Universidad Carlos III
Escuela Politécnica Superior

Servicio de Geoposicionamiento Cooperativo para Dispositivos Móviles

Autor: Francesco Gatti Gómez
Tutor: Jorge Blasco Alís

Mayo 2012

ÍNDICE

ÍNDICE DE CONTENIDO

0 ÍNDICE

0 ÍNDICE

0 ÍNDICE

ÍNDICE DE FIGURAS

0 ÍNDICE

1 INTRODUCCIÓN

1.1 OBJETIVO Y MOTIVACIÓN

La motivación de desarrollar este proyecto surge de una necesidad real. En determinados contextos un usuario puede estar interesado en compartir su ubicación con sus contactos. De esta idea surge este proyecto, en el que se desarrolla una aplicación para Android (extensible a otras plataformas móviles en el futuro) que muestra al usuario, sobre mapas personalizados, su propia ubicación y la de sus amigos, siempre que estos hayan decidido compartirla.

Google Latitude ofrece una funcionalidad similar a la perseguida pero tiene algunas carencias. Se ejecuta sobre Google Maps que, aun siendo personalizable y muy potente, no permite utilizar mapas diferentes a los proporcionados. No actualiza la posición de los contactos en tiempo real, sino sólo cuando se cumplen determinadas circunstancias. Esto hace que no sea útil en situaciones donde la tasa de refresco y la información actualizada es importante.

La aplicación debe:

- Permitir a los usuarios conocer su ubicación y la de sus contactos en mapas personalizados.
- Recibir actualizaciones frecuentes de las posiciones de los contactos.
- Ser fácil de utilizar, automatizando la mayor parte de procesos y requiriendo la mínima interacción con el usuario.
- Ser potente y flexible, y que pueda ser adaptada a un elevado número de contextos.

Partiendo de estas consideraciones, se desarrolla la aplicación que emplea un número elevado de las opciones que Android ofrece a los desarrolladores. Además está respaldada por una conexión al servidor, alojado en Google App Engine que describiremos más adelante.

1.2 GEOLOCALIZACIÓN Y CONECTIVIDAD

Las aplicaciones móviles pueden aprovechar todo el potencial que ofrecen los dispositivos. En el caso de este proyecto, vamos a centrarnos especialmente en dos: la posibilidad de los dispositivos de saber en qué lugar se encuentran –geolocalización- y la conexión de datos permanente de que disponen, que permite que los dispositivos se comuniquen entre ellos a través del servidor de forma casi constante.

La localización la proporcionan varios métodos:

- GPS
- WIFI
- Antenas GSM

Dado que nuestro objetivo es alcanzar la máxima precisión posible, vamos a utilizar todos los medios posibles, en detrimento de la batería. El GPS se empleará siempre que esté disponible y el resto de métodos en caso negativo. Está previsto desarrollar un sistema de localización *indoor* basado en WIFI dentro de este mismo proyecto, pero queda fuera del alcance de este documento.

La conectividad viene proporcionada por las diferentes conexiones de datos disponibles como 3D o EDGE o por conexiones WIFI.

Vamos a aprovechar estas características, además de otras que se detallarán a lo largo de este documento, para desarrollar nuestra aplicación que permitirá a los usuarios conocer en todo momento donde se encuentran y donde se encuentran sus amigos.

1.3 PROYECTO COLABORATIVO

Este proyecto se realiza en conjunción con otro cuyo autor es Fernando García Radigales. Ambos proyectos están estrechamente relacionados con la intención de obtener un producto de mayor envergadura.

El proyecto que describe este documento se centra en la realización de una aplicación para dispositivos móviles y del servidor que la respalda. La aplicación aprovecha las características de geolocalización que brindan los actuales dispositivos móviles. Estas características resultan muy útiles cuando el usuario se encuentra en exteriores, especialmente teniendo en cuenta la disponibilidad de utilizar el servicio GPS que disponen la mayoría de teléfonos actuales. Pero el propósito de esta aplicación va más allá y se pretende que sea igualmente útil en contextos que se desarrollan en espacios interiores.

Por ello se desarrolla el proyecto asociado que se basa en crear un sistema de posicionamiento en interiores utilizando redes WiFi. Actualmente los dispositivos ofrecen localización basadas en redes inalámbricas y de telefonía pero su precisión es muy baja y solo resulta útil en exteriores. El proyecto asociado está ideado para que en un determinado contexto se desplieguen una serie de puntos de acceso WiFi, o se aprovechen los ya existentes, y, partir de la intensidad con que el dispositivo recibe esas señales, determinar la ubicación con la máxima precisión posible.

Ambos proyectos se integran para completar una aplicación que permite ser instanciada en contextos tanto interiores como exteriores, y facilitar la localización del usuario y de sus contactos.

2 ANÁLISIS

En esta sección nos vamos a encargar de definir qué es lo que pretendemos construir. Como hemos dicho el objetivo es construir una aplicación de geolocalización que nos permita compartir nuestra ubicación con los contactos deseados.

2.1 DESCRIPCIÓN DEL PROYECTO

Esta sección está dedicada a la captura de requisitos, pero vamos a hacer un pequeño resumen de la funcionalidad que perseguimos. La función fundamental es la de mostrar la ubicación del usuario y de sus contactos, por lo que buena parte del análisis se basa en ella. Hemos establecido que la frecuencia de las actualizaciones es un factor crítico, para garantizar fiabilidad a los usuarios. Este es uno de los contratiempos que experimentan usuarios de servicios como Google Latitude, la falta de actualizaciones de contactos recientes. Otra característica fundamental es la de poder utilizar mapas personalizados. Google Maps ofrece una funcionalidad magnifica pero es posible que necesitemos características que no nos ofrece.

La aplicación debe ofrecer la capacidad al usuario de registrarse. Tras el registro, es necesario que el usuario pueda ver que contactos de su agenda están utilizando la aplicación y pueda asignar permisos a los mismos. Después podrá seleccionar el mapa y ver donde se encuentra él y sus amigos. Además tendrá acceso a una lista que mostrará información sobre los contactos.

El usuario puede indicar en todo momento si desea que el servicio de posicionamiento y envío de la posición al servidor este activado o no. Además queremos que la aplicación tenga un servicio de 'Ping', que permita al usuario consultar la posición de otro bajo demanda.

2.2 ALTERNATIVAS TECNOLÓGICAS

Las alternativas relativas a la elección de la tecnología móvil, en el 2011, se reducen principalmente a dos: iPhone y Android. Ambas cuentan con sus respectivos SDKs muy completos. Utilizan lenguajes de programación diferentes: Objective-C y Java respectivamente.

Android permite programar en cualquier sistema operativo, Windows, MacOS o Linux, mientras que para programar en iOS es necesario tener hardware Apple. En cuanto a las cuotas de inscripción, los desarrolladores iOS pagan $99 anualmente y los de Android $25 una única vez.

Por motivos principalmente económicos y por la facilidad que supone, la elección se decanta casi inmediatamente por Android. Aun así somos conscientes de que cualquier aplicación móvil de calidad tiene versiones al menos para estos dos sistemas, y por ello vamos a desarrollar todo el proyecto siempre teniendo en cuenta que en el futuro puede ser necesario desarrollar una versión para iPhone. Android no nos obliga a utilizar hardware Apple, por lo que los ordenadores que ya tenemos nos sirven.

Para el desarrollo de los servicios web necesarios para dar soporte a la aplicación, se han contemplado varias opciones: servlets Java, PHP, RubyOnRails… La alternativa elegida es Google App Engine [4], un framework de Google que resulta ser fácil de programar, en Java también . La principal ventaja de elegir GAE es que Google nos ofrece de forma gratuita sus servidores hasta unas determinadas cuotas, por lo que no tenemos que preocuparnos de alquilar servidores ni instalar nada en ellos. Además la integración con Android puede hacerse de una forma relativamente sencilla.

2.3 ANDROID

La plataforma Android es el producto del trabajo de la Open Handset Alliance [0] , un grupo de organizaciones que colaboran para construir mejores teléfonos móviles. El grupo, liderado por Google, incluye a los operadores móviles, fabricantes de dispositivos móviles, fabricantes de componentes, soluciones de software y proveedores de plataformas y empresas de comercialización. Desde el punto de vista de desarrollo de software, Android se encuentra a la vanguardia del desarrollo de código abierto. [1]

El primer dispositivo Android en el mercado fue el dispositivo G1 fabricado por HTC. El dispositivo llegó a estar disponible después de casi un año de especulaciones. Al publicarse la versión 1.0 del SDK de Android empezaron a surgir las primeras aplicaciones.

Para estimular la innovación, Google patrocinó dos "Android Developer Challenges" donde se repartieron premios por millones de dólares. Pocos meses después de la salida al mercado del G1, se publicó el Android Market (actualmente llamado Google Play), lo que permite a los usuarios navegar y descargar aplicaciones directamente a sus teléfonos.

Dado el rango de capacidades de Android, sería fácil confundirlo con un sistema operativo de escritorio. Android está estructurado en capas sobre una base del núcleo de Linux. El subsistema de interfaz de usuario incluye:

- Ventanas
- Vistas
- Widgets para representar elementos comunes, tales como cuadros de edición, listas y listas desplegables

Android incluye un navegador embebido basado en WebKit, el mismo que emplea Safari en dispositivos iPhone.

Los dispositivos con Android como sistema operativo cuentan con una amplia variedad de opciones de conectividad, incluyendo WiFi, Bluetooth y de datos inalámbricos (por ejemplo, GPRS, EDGE y 3G). Además cuenta con una fuerte integración de todos los servicios de Google,

como Gmail, Maps, Calendar, etc. Android permite acceder a servicios proporcionados por el hardware del teléfono como el GPS, acelerómetros o la cámara.

Las aplicaciones móviles han tenido que esforzarse para alcanzar y mantener el nivel de sus contrapartidas de escritorio. Android aborda el reto de proporcionar gráficos avanzados con soporte integrado para el 2-D y 3-D, incluyendo la biblioteca OpenGL. Android proporciona una forma de almacenamiento basada en software libre: SQLite. La Figura 2.1 muestra una vista simplificada de las capas de software de Android.

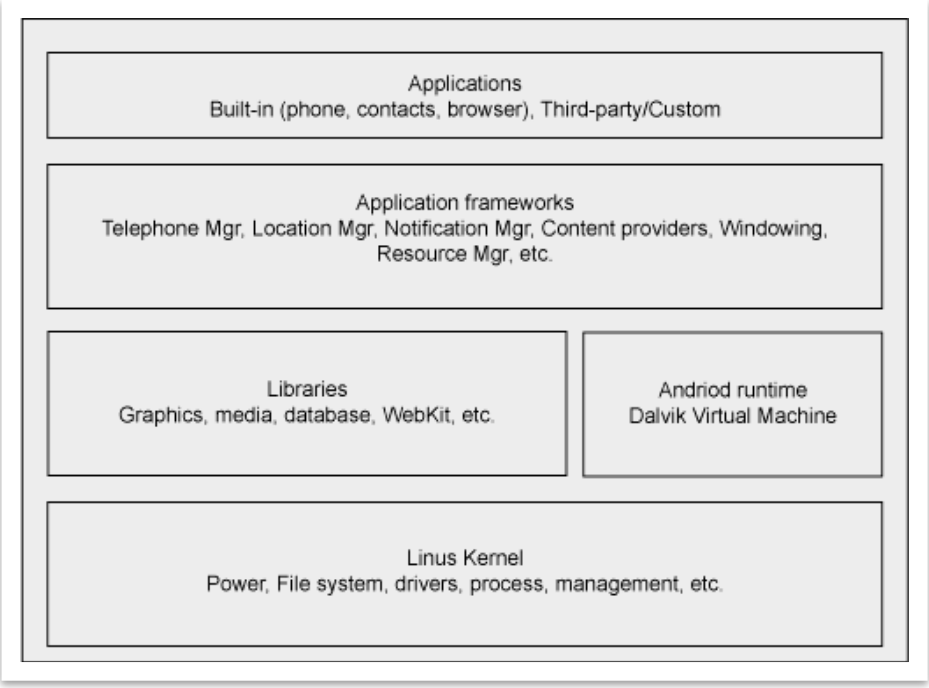

FIGURA 2.1 - ESTRUCTURA DE CAPAS DE ANDROID

2.4 GOOGLE APP ENGINE

App Engine es un servicio de alojamiento web que presta Google de forma gratuita hasta determinadas cuotas, este servicio permite ejecutar aplicaciones sobre la infraestructura de Google. Si no se cuenta con un dominio propio, Google proporciona uno con la siguiente estructura, midominio.appspot.com. También permite implementar un dominio propio a través de Google Apps. Por el momento las cuentas gratuitas tienen un límite de un gigabyte de almacenamiento permanente y la suficiente cantidad de ancho de banda y CPU para cinco millones de visitas mensuales, y si la aplicación supera estas cuotas, se pueden comprar cuotas adicionales. Las aplicaciones App Engine son fáciles de crear, de mantener y de ampliar al ir aumentando el tráfico y las necesidades de almacenamiento de datos. También permite limitar el acceso a determinados usuarios.

Actualmente las aplicaciones Google App Engine se implementan mediante los lenguajes de programación Python, Java y Go. [4]

Como hemos elegido desarrollar el backend en Java, éste estará basado en Servlets Java [5] y su implementación es similar. En nuestro caso, se definen una serie de entidades, que representan el estado de la aplicación y los usuarios, y servicios que permiten la comunicación con los clientes. La parte del servidor se encargará de controlar la parte de modelo, descrita en nuestra arquitectura.

2.4.1 C2DM

C2DM permite a terceros enviar mensajes ligeros a aplicaciones para Android. Se trata de un servicio de mensajería push que aprovecha la conexión de Google existente en los dispositivos. El servicio no está diseñado para enviar una gran cantidad de contenido a través de los mensajes. Por el contrario, se debe utilizar para indicar a la aplicación que hay nuevos datos en el servidor, de modo que la aplicación pueda tomar las medidas oportunas.

C2DM no ofrece ninguna garantía sobre la entrega o el orden de los mensajes. Por ejemplo en una aplicación de mensajería se usaría para indicar a la aplicación que el usuario tiene, mensajes nuevos, pero no enviaría el mensaje en si, por su falta de confirmación en la entrega.

Una aplicación Android no necesita estar en ejecución para recibir los mensajes C2DM. El sistema activará la aplicación a través de un Intent cuando el mensaje llegue, siempre y cuando la aplicación está configurada con el receptor adecuado y permisos.

No existe ninguna interfaz para manipular los datos del mensaje. C2DM simplemente pasa los datos en bruto recibidos directamente a la aplicación, que tiene control total para manejar la situación. Por ejemplo, la aplicación puede enviar una notificación, mostrar una interfaz de usuario personalizada, o sincronizar datos.

Se requiere que los dispositivos integren versiones de Android 2.2 o superiores.

C2DM utiliza la conexión existente de Google, por lo que es necesario estar autentificado con una cuenta Google para poder utilizarlo. Es el mismo servicio de mensajes push que utilizan aplicaciones como GMail o Google Maps.

En nuestra aplicación, este componente esta implementado como una clase en Android que se encarga de recibir las notificaciones. Desde Google App Engine se pueden enviar mensajes C2DM simplemente llamado a un método estático

2.4.2 GOOGLE WEB TOOLKIT

GWT o Google Web Toolkit es un framework creado por Google que permite ocultar la complejidad de varios aspectos de la tecnología AJAX. Es compatible con varios navegadores, lo cual es notorio ya que cada navegador suele necesitar código específico para lograr un front-end correcto en una aplicación web. El concepto de Google Web Toolkit es bastante sencillo, básicamente lo que se debe hacer es crear el código en Java usando cualquier IDE de Java y el compilador lo traducirá a HTML y JavaScript.

Con la biblioteca GWT, los desarrolladores pueden crear y depurar aplicaciones AJAX en lenguaje JAVA usando el entorno de desarrollo que prefieran. Cuando una aplicación es desplegada, el compilador GWT traduce la aplicación Java a un archivo Javascript, que puede ser ofuscado para optimizar el rendimiento.

GWT no es sólo una interfaz de programación; proporciona un conjunto de herramientas que permiten desarrollar funcionalidades Javascript de alto rendimiento en el navegador del cliente.

Una aplicación GWT puede ser ejecutada en dos modos:

- Modo host (Hosted mode): La aplicación se ejecuta como código bytecode de Java dentro de la Máquina Virtual de Java (JVM). Este modo es el más usado para desarrollo, soportando el cambio de código en caliente y el depurado.
- Modo web (Web mode): La aplicación se ejecuta como código Javascript y HTML puro, compilado a partir del código Java. Este modo se suele usar para el despliegue de la aplicación.

Aunque en esta aplicación no existe ninguna aplicación web, sí que utilizamos algunas funciones que vienen incluidas en GWT, como por ejemplo el sistema de llamadas remotas RequestFactory. [8]

2.5 REPRESENTACIÓN EN EL MAPA

La función principal de la aplicación es representar sobre un plano la posición del usuario y de sus contactos. Para tal propósito, existen varias alternativas. La más sencilla seria utilizar Google Maps y mostrar los elementos deseados sobre él. La integración de Google Maps con Android hace muy sencillo realizar esta tarea. Pero finalmente no se ha elegido esta opción, aunque no se descarta que pueda ser implementada de forma complementaria en el futuro.

La opción elegida consiste en replicar parte del comportamiento de Google Maps y Google Latitude pero adaptándolo a nuestras necesidades. Nuestra aplicación se utiliza en contextos acotados, como el campus de una universidad o una estación de esquí. Por ello como plano podemos utilizar una imagen png o jpg y desplegar los iconos que queramos sobre ella. De esta manera no estamos atados a los mapas de Google Maps y ganamos en flexibilidad para poder crear mapas a nuestras necesidades. Tanto es así, que el sistema que hemos creado para representar los mapas nos permite utilizar planos que no tengan correspondencia escalar

directa con la realidad. Es decir, no hay una escala que podamos utilizar en todo el mapa y las distancias y direcciones entre puntos no tienen correspondencia directa con la realidad.

Aun así necesitamos otras funciones que si incluye Google Maps, como poder desplazarnos por el mapa moviendo un dedo, o acercar o alejar el zoom haciendo el característico gesto de unir o separar dos dedos. Todo esto debe suceder de forma que los iconos representados sobre el mapa que marcan nuestra posición y la de nuestros contactos se desplacen de manera correcta con el plano y siempre marquen la misma posición, no importa el nivel de zoom o el desplazamiento. Se explican más detalles de cómo se ha creado la vista en la sección Vistas Personalizadas.

Como hemos dicho, hemos querido crear unos mapas tan flexibles que nos permitan representar mapas que no tienen necesariamente una relación directa con la realidad. Vamos a poner un ejemplo.

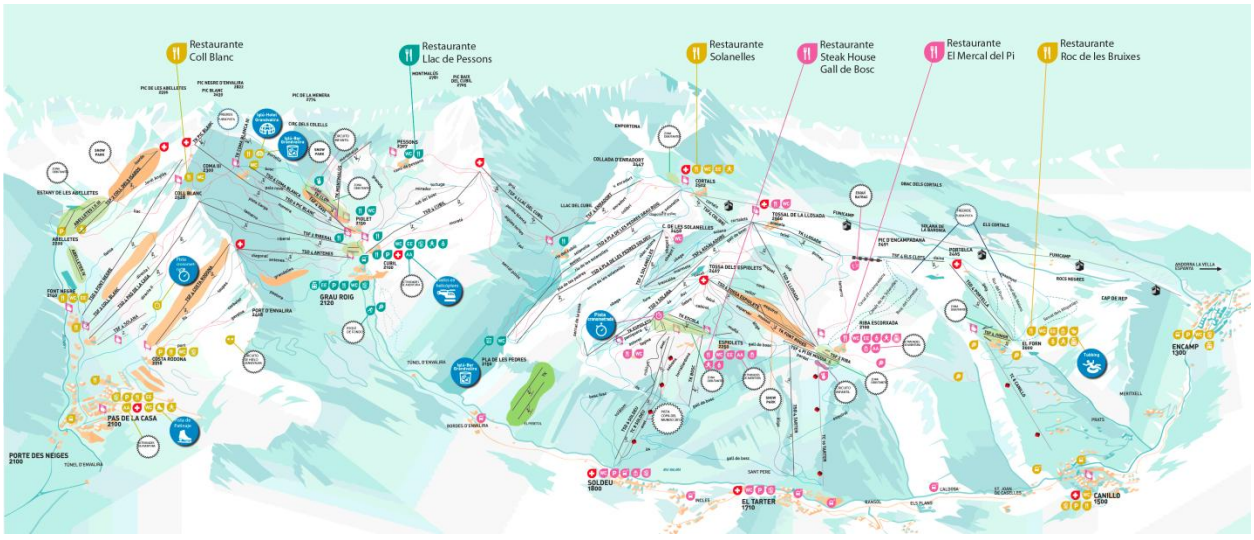

FIGURA 2.2 - PLANO GRAND VALIRA

En la figura 2.2 podemos ver el mapa real que la estación de esquí Grand Valira en Andorra ofrece a sus clientes para orientarse en la misma. Este mapa representa una vista similar a la que obtendríamos si nos colocásemos en lo alto de una montaña en frente de la estación. Solo que en realidad no es así, las distancias entre montañas, remontes, pistas o edificios no

corresponde con la realidad y se dan situaciones como que nos deja "ver" por detrás de algunas montañas.

Situaciones como esta nos generan un problema. Por un lado utilizar este mapa en este contexto resulta perfecto, porque para los usuarios es un mapa familiar, característico de estaciones de esquí y fácil de interpretar. Por otro la falta de correspondencia con distancias reales dificulta la tarea.

2.6 REQUISITOS

Esta sección se compone de una serie de requisitos de usuario, que usaremos como referencia para el posterior diseño y desarrollo de la aplicación. El formato que utilizaremos para describir cada requisito en esta sección es el siguiente:

ID	**Identificador del Requisito**			**Tipo**	**Funcional o No Funcional**
Nombre	Nombre breve y descriptivo del requisito				
Actores	Actores involucrados en el cumplimiento del requisito				
Descripción	El usuario debe poder registrarse en la aplicación. Las formas de identificación son el número de teléfono o el email.				
Objetivo	Objetivo que se persigue con el requisito, puede ser una Capacidad o una Restricción.				
Verificable	Si o No	**Esencial**	Si o No	**Prioridad**	0 - 5

Separamos la lista de requisitos entre funcionales y no funcionales

Los valores del campo Prioridad oscilan de 0 a 5, representando 0 la prioridad mínima, y 5 la prioridad máxima.

2.6.1 FUNCIONALES

ID	UR-01			Tipo	Funcional
Nombre	Registro				
Actores	Usuario				
Descripción	El usuario debe poder registrarse en la aplicación. Las formas de identificación son el número de teléfono y el email. Es necesario que el usuario tenga una cuenta de Google asociada al dispositivo y que introduzca su número de teléfono				
Objetivo	Capacidad				
Verificable	Si	Esencial	Si	Prioridad	5

ID	UR-02			Tipo	Funcional
Nombre	Login				
Actores	Usuario				
Descripción	El usuario debe ser capaz de autentificarse. Se tratará de un proceso automático que se realizará al abrir la aplicación utilizando la última información de registro. El usuario no debe hacer nada. En cambio puede hacer logout, dando al usuario de nuevo la oportunidad de registrarse con una cuenta de correo y teléfono diferentes.				
Objetivo	Capacidad				
Verificable	Si	Esencial	Si	Prioridad	5

ID	UR-03			Tipo	Funcional
Nombre	Búsqueda de contactos				
Actores	Usuario				
Descripción	El usuario debe poder localizar entre los contactos de su agenda a los individuos que ya estén utilizando la aplicación, basándose en su dirección de correo electrónico o su teléfono. Este proceso debe ejecutarse de forma automática y en segundo plano.				
Objetivo	Capacidad				
Verificable	Si	Esencial	Si	Prioridad	5

ID	UR-04			Tipo	Funcional
Nombre	Lista de usuarios				
Actores	Usuario				
Descripción	Los amigos localizados deben ser mostrados en una lista, donde se provee con información adicional al nombre, como ubicación, última actualización de estado, distancia y foto.				
Objetivo	Capacidad				
Verificable	Si	Esencial	Si	Prioridad	4

ID	UR-05		Tipo	Funcional
Nombre	Permisos			
Actores	Usuario			
Descripción	El usuario debe ser capaz de decidir a qué contactos se le concede permiso para conocer su ubicación.			
Objetivo	Capacidad			
Verificable	Si	Esencial	Si	Prioridad 3

ID	UR-06		Tipo	Funcional
Nombre	Representación del mapa			
Actores	Usuario			
Descripción	Es necesario que exista una representación del contexto elegido en forma de mapa, donde aparezcan tanto el usuario como los contactos. El usuario debe tener un identificador característico (de tipo pin) y los contactos vendrán representados por su foto.			
Objetivo	Capacidad			
Verificable	Si	Esencial	Si	Prioridad 4

ID	UR-07		Tipo	Funcional	
Nombre	Funcionalidad del mapa				
Actores	Usuario				
Descripción	El mapa debe tener capacidad para hacer zoom y desplazarse, utilizando los gestos habituales con los dedos en dispositivos móviles. La aplicación puede utilizar mapas ya incluidos en el paquete de instalación o permitir descargar mapas desde el servidor bajo demanda.				
Objetivo	Capacidad				
Verificable	Si	Esencial	Si	Prioridad	3

ID	UR-08		Tipo	Funcional	
Nombre	Botón de posicionamiento				
Actores	Usuario				
Descripción	Tiene que existir un botón, claramente visible, que permita al usuario activar o desactivar el posicionamiento en segundo plano.				
Objetivo	Capacidad				
Verificable	Si	Esencial	Si	Prioridad	3

ID	UR-09			Tipo	Funcional
Nombre	Función 'Ping'				
Actores	Usuario				
Descripción	El usuario tiene la capacidad de hacer 'Ping' a otro usuario, solicitándole que indique la posición. El usuario receptor indicará su posición automáticamente, en caso de haber concedido permiso para conocer su posicion al usuario emisor.				
Objetivo	Capacidad				
Verificable	Si	Esencial	Si	Prioridad	4

ID	UR-10			Tipo	Funcional
Nombre	Interfaz				
Actores	Usuario				
Descripción	La interfaz tendrá una disposición en pestañas, e incluirá la lista de amigos y la representación del mapa. La interfaz debe ser lo suficientemente flexible para que resulte sencillo añadir pestañas, en el caso en que el contexto en que se instancia la aplicación lo requiera.				
Objetivo	Capacidad				
Verificable	Si	Esencial	Si	Prioridad	3

ID	UR-11			Tipo	Funcional
Nombre	Forzar búsqueda de contactos				
Actores	Usuario				
Descripción	El usuario puede solicitar manualmente una nueva búsqueda de usuarios entre los contactos de la agenda.				
Objetivo	Capacidad				
Verificable	Si	Esencial	No	Prioridad	3

ID	UR-12			Tipo	Funcional
Nombre	Notificaciones				
Actores	Usuario				
Descripción	El usuario debe recibir notificaciones en los siguientes casos: - Se activa el servicio de posicionamiento - Se recibe un 'Ping'				
Objetivo	Capacidad				
Verificable	Si	Esencial	No	Prioridad	3

2.6.2 NO FUNCIONALES

ID	UR-13			Tipo	No Funcional
Nombre	Servidor				
Actores	Sistema				
Descripción	La aplicación debe estar respaldada por un servidor encargado de gestionar el registro, login, permisos y distribución de posiciones.				
Objetivo	Capacidad				
Verificable	Si	Esencial	Si	Prioridad	4

ID	UR-14			Tipo	No Funcional
Nombre	Datos agenda Android				
Actores	Sistema				
Descripción	La información que se mostrará para cada contacto será extraída de la agenda incluyendo nombre del contacto y foto.				
Objetivo	Restricción				
Verificable	Si	Esencial	Si	Prioridad	4

ID	UR-15			Tipo	No Funcional
Nombre	Compartición de posiciones				
Actores	Sistema				
Descripción	La aplicación debe ser capaz de enviar a los usuarios las posiciones de los contactos, en tiempo semi-real. Se considerará tiempo semi-real a intervalos de actualización de menos de un minuto. Este valor debe ser configurable.				
Objetivo	Restricción				
Verificable	Si	Esencial	Si	Prioridad	4

ID	UR-16			Tipo	No Funcional
Nombre	Mapas personalizados				
Actores	Sistema				
Descripción	Los mapas deben utilizar como base imágenes personalizables y tener la capacidad de mostrar elementos sobre las mismas. Los mapas se compondrán de una imagen base sobre la que se superponen la posicion del usuario y de los contactos.				
Objetivo	Capacidad				
Verificable	Si	Esencial	Si	Prioridad	4

ID	UR-17			Tipo	No Funcional
Nombre	Servicio de posicionamiento				
Actores	Sistema				
Descripción	El servicio de posicionamiento se ejecuta en segundo plano. Se encarga de recibir notificaciones periódicas del servidor y enviarlas al servidor.				
Objetivo	Restricción				
Verificable	Si	Esencial	Si	Prioridad	4

ID	UR-18			Tipo	No Funcional
Nombre	Sincronización de permisos				
Actores	Sistema				
Descripción	Debe existir consistencia entre el estado de los permisos en el cliente y en el servidor.				
Objetivo	Restricción				
Verificable	Si	Esencial	Si	Prioridad	3

ID	UR-19			Tipo	No Funcional
Nombre	Conversión de coordenadas				
Actores	Sistema				
Descripción	La aplicación debe ser capaz de convertir coordenadas geográficas al sistema utilizado por los mapas.				
Objetivo	Restricción				
Verificable	Si	Esencial	Si	Prioridad	4

3 DISEÑO

3.1 INTRODUCCIÓN

Concluida la captura de requisitos se detalla la arquitectura del sistema.

Vamos a utilizar un patrón de diseño bien conocido, el patrón modelo-vista-controlador. Para el diseño de aplicaciones con sofisticados interfaces se utiliza el patrón de diseño Modelo-Vista-Controlador. La lógica de un interfaz de usuario cambia con más frecuencia que los almacenes de datos y la lógica de negocio. Si no separamos correctamente estas partes, cambios en la interfaz pueden resultar costosos por obligarnos a cambiar código que no deberíamos.

Se trata de realizar un diseño que desacople la vista del modelo, con la finalidad de mejorar la reusabilidad. De esta forma las modificaciones en las vistas impactan en menor medida en la lógica de negocio o de datos.

Elementos del patrón:

- Modelo: datos y reglas de negocio
- Vista: muestra la información del modelo al usuario
- Controlador: gestiona las entradas del usuario

El **modelo** es el responsable de:

- Acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.

- Define las reglas de negocio (la funcionalidad del sistema).
- Lleva un registro de las vistas y controladores del sistema.

El **controlador** es responsable de:

- Recibir los eventos de entrada (una pulsación sobre la pantalla del dispositivo, un cambio en un campo de texto, etc.).
- Contener reglas de gestión de eventos, del tipo "SI Evento Z, entonces Acción W". Estas acciones pueden suponer peticiones al modelo o a las vistas.

Las **vistas** son responsables de:

- Recibir datos del modelo y mostrarlos al usuario.
- Tienen un registro de su controlador asociado (normalmente porque además lo instancia).
- Pueden ofrecer el servicio de Actualización(), para que sea invocado por el controlador o por el modelo (cuando es un modelo activo que informa de los cambios en los datos producidos por otros agentes).

En nuestro caso, la vista y el controlador se desarrollan por completo en el cliente Android, mientras que el modelo se basará en un servicio web, creado con Google App Engine.

En las siguientes secciones se dan más detalles.

En el apartado de diseño definiremos una serie de componentes, encargados de satisfacer los requisitos recabados en la sección anterior.

3.2 ANDROID

Una parte muy importante de este proyecto consiste en el desarrollo del cliente Android. El objetivo es crear una interfaz robusta y sencilla. Se pretende que para el usuario resulte fácil el uso de la misma. Se automatizan todos los procesos posibles de forma que sean transparentes para el mismo y solo se requiera su intervención en caso de ser imprescindible.

Para realizar el diseño del cliente vamos a tener en cuenta la arquitectura específica de Android. Cada aplicación en Android corre en un proceso principal, que está dedicado a actualizar la UI. Todo procesamiento que no tenga este fin debe ejecutarse en procesos ligeros secundarios. Android provee de varios sistemas para tal fin, como la clase ASyncTask o los servicios.

3.2.1 DIAGRAMA DE CLASES

Vamos a diseñar un diagrama de alto nivel de las clases. En esta sección solo vamos a centrarnos en el cliente Android, que incluye la vista y el controlador.

La vista se localiza en las clases Android que representan actividades, encontrándose todas en el paquete Actividades. El controlador se encuentra en el resto de paquetes. Mención especial requiere el paquete almacenamiento. Aunque hemos establecido que el modelo se encuentra en el servidor, los dispositivos Android dependen de conexiones de red que pueden no estar siempre disponibles. Por ello necesitamos almacenar determinados datos en el dispositivo, a modo de caché.

3 DISEÑO

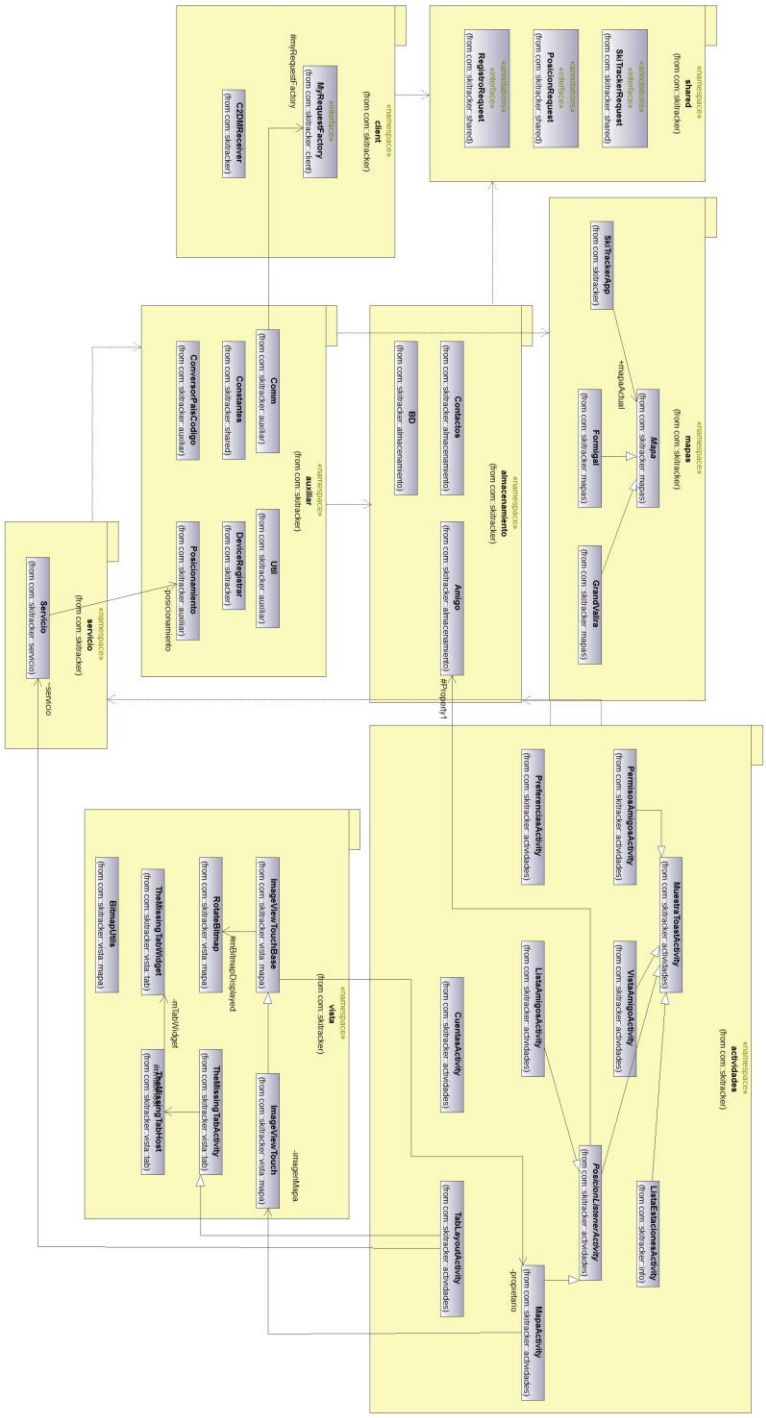

FIGURA 3.1 - DIAGRAMA DE CLASES ANDROID

3.2.2 ACTIVIDADES

Como hemos dicho, se persigue que la aplicación sea lo más fácil de usar posible. Por ello el número de actividades es limitado y pasamos a describir cada una a continuación.

3.2.2.1 CO-01: TABLAYOUTACTIVITY

Se trata de la actividad principal, cuya función es contener a las demás y proporcionar un sistema de pestañas para facilitar la navegación.

3.2.2.2 CO-02: CUENTASACTIVITY

El objetivo de esta actividad es proporcionar una forma de registro. La actividad dispone de 2 *layouts* diferentes, uno se muestra en caso de que la aplicación no se haya validado con el App Engine y otra en caso de que sí. En el primer caso se muestra la lista de cuentas de Google disponibles para la autentificación. En el segundo se muestra el nombre de la cuenta con que se está autentificado y se da la opción de desconectar. Podría darse el caso en que no hubiese ninguna cuenta de Google asociada al dispositivo (aunque infrecuente). De ser así se llama a una actividad del sistema mediante un Intent que permite asociar o crear una nueva cuenta.

3.2.2.3 CO-03: POSICIONLISTENERACTIVITY

Se trata de una actividad abstracta de la que heredan las siguientes dos: ListaAmigosActivity y MapaActivity. Su objetivo es doble, por un lado recibe las nuevas posiciones del servicio y por otro consulta al servidor periódicamente las nuevas posiciones de los contactos. Esta actividad es abstracta y obliga a las clases herederas implementar métodos cuando se recibe una nueva posición propia o de los contactos.

3.2.2.4 CO-04: LISTAAMIGOSACTIVITY

Esta actividad hereda de la anterior y su objetivo es mostrar una lista de los contactos que utilizan la aplicación. Si uno de estos contactos está en uno de los entornos definidos por el contexto de la aplicación, se dará información al usuario de la localización del mismo. Por ejemplo, si suponemos que estamos en el entorno de la Universidad Carlos III, y uno de los contactos está en el recinto, la información que aparecerá podría ser como en la figura 3.1.

La lista que aparece en el dispositivo es de los usuarios encontrados en la agenda que utilizan la aplicación. El proceso de búsqueda de amigos se realiza automáticamente y se detalla en la sección Procesos Automáticos. La imagen que aparece al lado de cada usuario es la que tiene en la agenda, al igual que el nombre. En caso de que no se encuentre la imagen se emplea una imagen genérica, y, si no se encuentra el nombre de usuario, se emplea el email del mismo.

FIGURA 3.1 - LISTAAMIGOSACTIVITY

Se incluye dentro de TabLayoutActivity e incluye un botón grande en la base para acceder a los permisos. En caso de hacer click en un amigo, se abre en un dialogo la actividad VistaAmigoActivity, que se detallará a continuación.

Si accedemos al menú desde esta actividad podemos:

- Establecer permisos
- Acceder a la actividad CuentasActivity
- Buscar nuevos amigos que hayan empezado a utilizar la aplicación
- Acceder a la actividad de configuración

3.2.2.5 CO-05: MAPAACTIVITY

Como en el caso anterior, esta clase también hereda de PosicionListenerActivity por lo que escucha los cambios en la posición propia y la de los contactos. Esta actividad detecta automáticamente si el usuario se encuentra en algún recinto acotado por un mapa y de ser así lo abre automáticamente. En caso negativo se muestra un *layout* con los mapas disponibles. Una vez seleccionado uno se muestra la vista del mapa, y sobre ella la posición del usuario y de los contactos, de encontrarse en él.

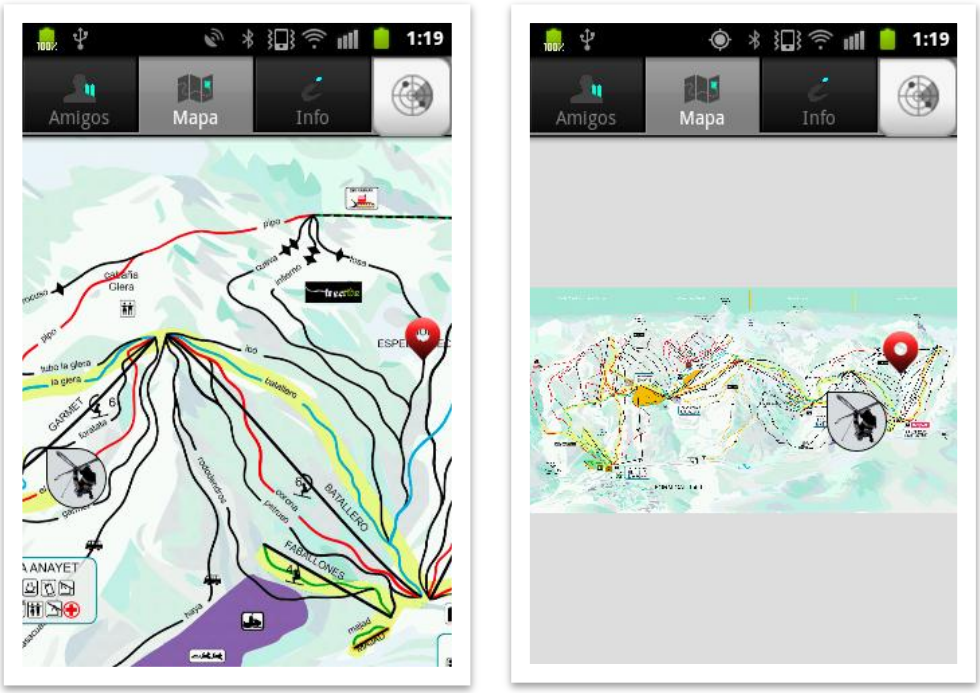

FIGURA 3.2 - MAPAACTIVITY VERTICAL

De la misma manera que en la actividad anterior, si hacemos click largo sobre uno de los amigos en la vista de mapa se abre la actividad VistaAmigoActivity.

El menú de esta actividad permite consultar al servidor la última posición de los amigos y cambiar de mapa.

En las figuras 3.2 y 3.3 se puede ver la actividad mapa en modo apaisado y vertical.

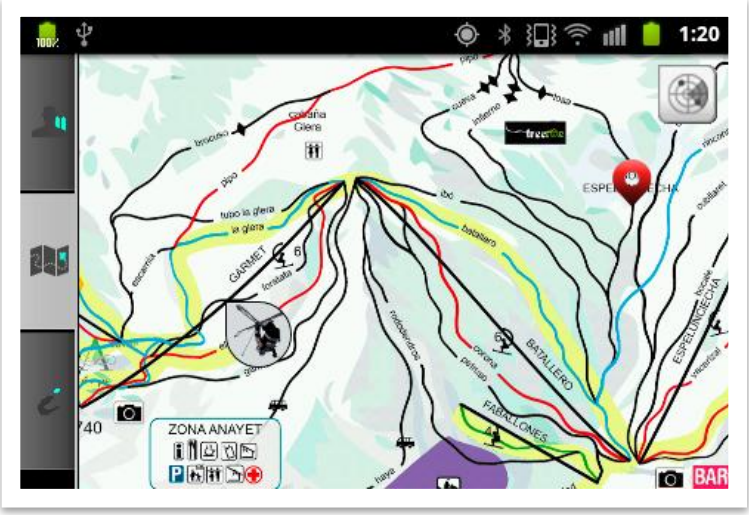

FIGURA 3.3 - MAPAACTIVITY APAISADO

3.2.2.6 CO-06: VISTAAMIGOACTIVITY

Esta actividad se abre siempre como un dialogo flotante sobre la actividad que la llama, siempre una de las dos anteriores. Puede observarse en la figura 3.5.

Proporciona información básica, nombre de usuario, localización y tiempo de la misma y una foto. Además permite hacer “Ping” al usuario, que implica pedirle que actualice su posición. Se detalla esta funcionalidad en el apartado de comunicaciones.

FIGURA 3.4 - VISTA DE CONTACTO

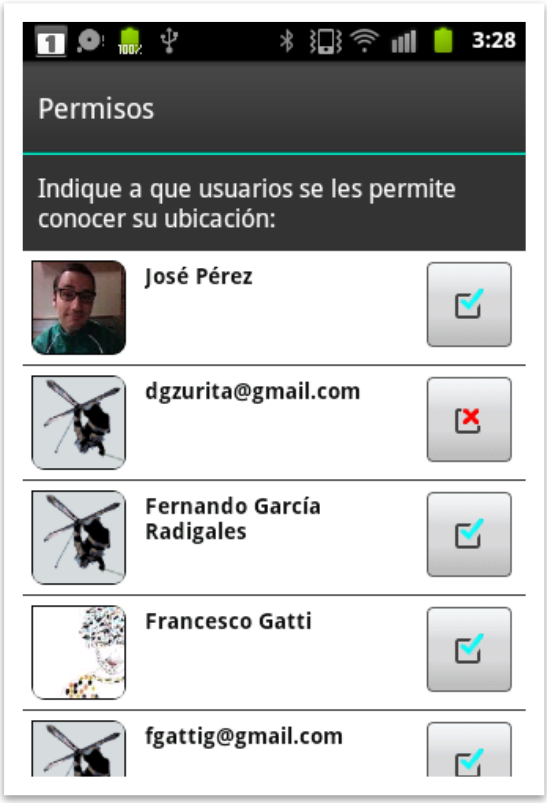

FIGURA 3.5 - PERMISOSAMIGOSACTIVITY

3.2.2.7 CO-07: PERMISOSAMIGOSACTIVITY

Esta actividad se encarga de mostrar una lista con los contactos que usan la aplicación y permitir asignar o revocar el permiso de ver la ubicación propia a cada uno de ellos.

3.2.2.8 CO-08: PREFERENCIASACTIVITY

Esta actividad sigue el patrón estándar de menú de preferencias de Android y tiene únicamente una opción de selección, la de activar el posicionamiento de forma automática al iniciar la aplicación.

Se accede a ella desde el menú.

3.2.3 ALMACENAMIENTO

Aunque la aplicación tiene una fuerte dependencia de la conexión de datos, ya que la información que maneja conviene que esté lo más actualizada posible, es necesario almacenar determinados datos en caso de que no esté disponible. También sirve para reducir el número de conexiones y cargar inmediatamente datos como la última posición de los amigos conocida. El almacenamiento se realiza tanto mediante base de datos como en ficheros en almacenamiento externo.

Por lo dicho hasta ahora, este paquete no se encuadra en la parte del controlador, sino más bien en el modelo y es la excepción en diseño que hemos establecido.

3.2.3.1 CO-09: SQLITE

Android pone una potente y a la vez ligera base de datos al servicio del desarrollador. Lo único que hay que hacer es extender la clase `SQLiteOpenHelper`, estableciendo métodos para crear, actualizar y acceder a la base de datos. En nuestro caso, tenemos la clase `BD.java` que se encarga de gestionar objetos de tipo `Amigo`.

La función de este componente es la de asegurar la persistencia del estado de los contactos entre ejecuciones de la aplicación y durante el desarrollo de la misma. Otros componentes pueden consultar en cualquier momento a este, obteniendo una lista de amigos donde se incluye información como el nombre, el ID de Android, última vez que se le vio, email y si tiene o no permiso. Del mismo modo, el resto de componentes pueden solicitar realizar cambios en la base de datos, por ejemplo si recibimos del servidor una nueva posición de un contacto.

3.2.3.2 CO-10: ALMACENAMIENTO EXTERNO

Además de la base de datos, se utiliza un recurso más, consistente en guardar determinados objetos en la tarjeta de memoria. Es importante que exista consistencia entre en el estado del servidor y el estado del cliente. Hay un proceso automático que describiremos más adelante cuyo objetivo tiene buscar los teléfonos y emails de los contactos en la agenda y comprobar en el servidor que usuarios están incluidos. Para evitar estar enviando constantemente toda la agenda de teléfonos al servidor, proceso costoso tanto en procesamiento como en

comunicaciones, cuando un email o teléfono se ha comprobado se almacena en la tarjeta de memoria para evitar comprobarlo en el futuro.

Para este fin existe la clase `Contactos` de tipo serializable, que incluye un set de teléfonos y otro de emails. Esta clase proporciona métodos estáticos para:

- Crear un objeto Contactos a partir del contenido de la agenda: se leen todos los contactos de la agenda y se incluyen los emails y teléfonos en sus respectivos Sets.
- Guardar un objeto Contactos en almacenamiento externo
- Recuperar un objeto Contactos previamente guardado.

El uso de esta clase se detallará la sección de Procesos Automáticos.

3.2.4 SERVICIO Y POSICIONAMIENTO

Android proporciona varios mecanismos para ejecutar procesos en segundo plano. En caso de procesos que se ejecutan indefinidamente, incluso cuando la aplicación no está en primer plano, Android recomienda el uso de un servicio.

3.2.4.1 CO-11: POSICIONAMIENTO

En nuestro caso este proceso de duración indefinida se encarga de detectar los cambios de posición y actuar en consecuencia. Para ello se apoya en una clase llamada `Posicionamiento`, que hereda de LocationListener. Vamos a describir por separado el funcionamiento de ambas clases.

3.2.4.2 CO-12: SERVICIO

La clase Servicio hereda lógicamente de la clase `Service`. Como se describe en el apartado Interfaz, existe un botón grande que sirve para dejar activado el posicionamiento una vez se sale de la aplicación. De esta manera nuestra posición se sigue enviando periódicamente al servidor y nuestros contactos pueden saber dónde estamos. Por lo tanto nos interesa que el servicio esté activado cuando se ha pulsado este botón pero también cuando la aplicación está en primer plano, ya que hay varias actividades que escuchan nuestros cambios de posición. La actividad principal y la clase Servicio son los que se encargan de gestionar el ciclo del vida del último.

3.2.4.3 FUNCIONAMIENTO

El servicio se activa cuando la aplicación está en primer plano o cuando no lo está pero el usuario ha indicado que quiere mantener el posicionamiento activado. El funcionamiento normal es el siguiente:

- Se activa el servicio y por lo tanto el posicionamiento
- El proveedor de Localización, que bien puede ser GPS, WIFI o celdas GSM, nos envía una nueva posición.
- Si la aplicación está en primer plano, hacemos un Broadcast Intent para enviarle la posición y que se encargue de recibirlo la actividad activa en ese momento.

- Enviamos la posición al servidor para que pueda ser compartida con los contactos que hemos elegido como permitidos.

Lo ideal sería hacer esto constantemente, pero por estar desarrollando para un dispositivo móvil donde la batería es un recurso preciado, no podemos. Cuando el servicio está activado y la actividad no está en primer plano, el posicionamiento solo se activa a intervalos regulares y no constantemente. Así podemos apagar el GPS buena parte del tiempo y ahorrar batería. Cuando recibimos una nueva posición, la comparamos con la anterior en términos de precisión y de hora, y si la consideramos mejor, desactivamos el posicionamiento hasta la siguiente actualización prevista. Por otro lado, tampoco se puede enviar al servidor todas las posiciones que recibimos y solo se envían si ha pasado más de un determinado tiempo desde el ultimo envío. En este caso la tasa de envíos es personalizable pero se recomienda que sea del orden de uno o más minutos.

Cuando la aplicación está en primer plano la tasa de refresco es mayor que en el caso anterior, pero solo a la hora de enviar Intents para actualizar la interfaz, es decir no varía la tasa en que se envía la posición al servidor.

Para dejar claro la separación de funciones:

- El servicio se encarga de gestionar su propio ciclo de vida y activar el posicionamiento. Además manda órdenes a la clase posicionamiento en función de qué modo se encuentre.
- La clase posicionamiento se encarga de recibir las posiciones, retrasmitirlas a la interfaz y enviarlas al servidor gestionando las tasas de cada evento. También se desactiva automáticamente cuando la aplicación no está en primer plano y no se esperan nuevas posiciones en un periodo de tiempo. Puede darse el caso que si deseemos recibir una actualización pero no recibamos ninguna durante un periodo de tiempo, por ejemplo si estamos en el interior de un edificio y solo tenemos disponible el GPS. De ser así, se anula el intento de posicionamiento y se suspende hasta la próxima actualización deseada. De no hacer esto, el GPS estaría constantemente activado en interiores suponiendo un fuerte gasto de batería, a la par que inútil.

3 DISEÑO

La combinación Servicio/Posicionamiento cumple con una función más. La aplicación da la opción de hacer “Ping” para conocer la posición de un contacto. Puede darse el caso que la aplicación no esté en primer plano y el usuario no haya activado el posicionamiento. Si esto ocurre, al recibir Ping, se activa el servicio y el posicionamiento en segundo plano y se escuchan posiciones hasta que recibir una valida, que es enviada al servidor. Después de esto se desactiva todo y se continúa como anteriormente.

La clase Servicio también es la encargada de mostrar u ocultar la notificación en la barra de estado cuando el usuario pulsa el botón de posicionamiento.

3.2.4.4 DIAGRAMA

A continuación se incluye un diagrama de clases del paquete, donde se incluyen los métodos de cada una.

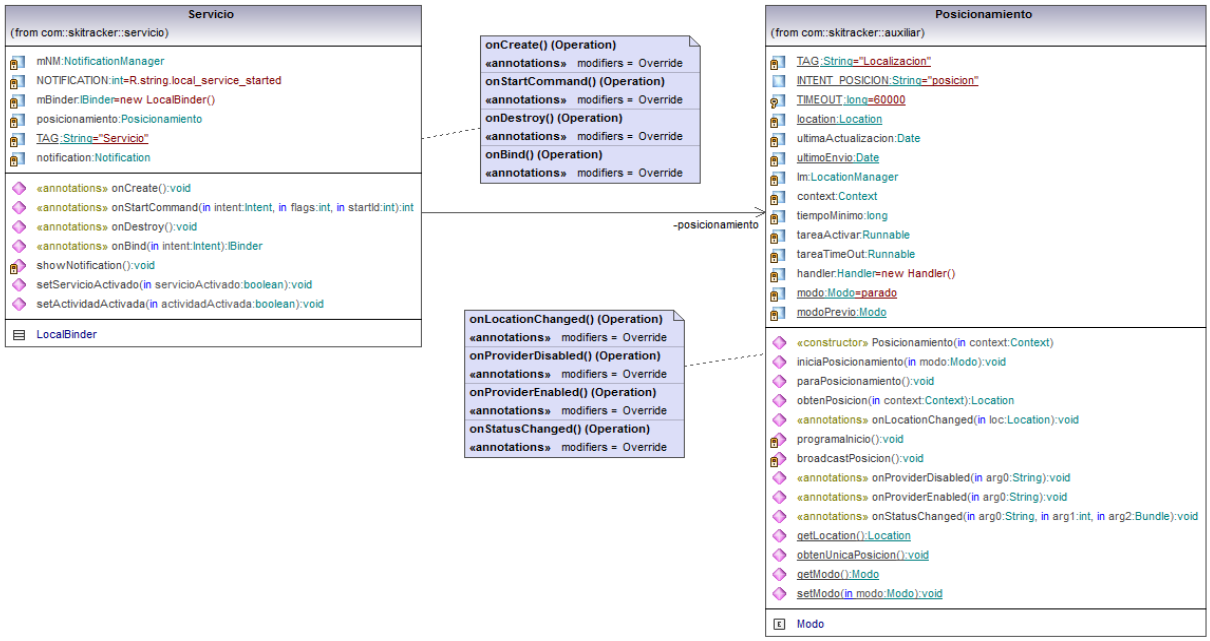

FIGURA 3.6 - DIAGRAMA DE SERVICIO Y POSICIONAMIENTO

3.2.5 PROCESOS AUTOMÁTICOS

Uno de los objetivos planteados al iniciar este proyecto ha sido buscar la máxima facilidad de uso y tratar de lograr la mínima interacción con el usuario, requiriendo su atención solo en caso imprescindible.

3.2.5.1 CO-13: BÚSQUEDA AUTOMÁTICA DE CONTACTOS

Dado que tenemos acceso a la agenda del usuario, podemos hacer que el proceso de buscar amigos sea automático. El objetivo es que al iniciar la aplicación la lista de contactos se actualice espontáneamente sin que el usuario tenga que buscar en la agenda o introducir emails manualmente o conectarse a algún servicio externo como una red social. La aplicación requiere que cada usuario este autentificado con una cuenta de Google y además se conoce el número de teléfono de cada usuario. Para que la lista esté siempre actualizada se ejecutan los siguientes procesos:

1. Al registrarse en la aplicación se genera un objeto `Contactos`, explicado en el apartado Almacenamiento, que incluye todos los emails y teléfonos encontrados en la agenda. Este objeto es serializado y guardado en memoria. La lista de contactos se envía al servidor y este devuelve otra con los contactos encontrados que ya usan la aplicación.
2. Al iniciar la aplicación en veces sucesivas, ya sabemos que contactos hemos comprobado con el servidor, así que miramos en la agenda y en el fichero `Contactos` que hemos guardado previamente y consultamos solo sobre los nuevos contactos aparecidos. Así nos aseguramos que al iniciar la aplicación se han comprobado todos los contactos.
3. Podría darse el caso que un usuario comenzase a utilizar la aplicación después de que otro haya comprobado su entrada de la agenda con el servidor. Para evitar inconsistencias, cuando un usuario se registra y envía su agenda al servidor, se notifica por C2DM a todos los usuarios encontrados que otro usuario les tiene como amigos. Es decir, cuando un usuario se registra y se encuentran amigos, a estos amigos también les aparecerá el nuevo usuario en la lista de contactos.
4. En caso de que estemos utilizando la aplicación y modifiquemos la agenda para añadir un nuevo contacto (no se cumpliría la segunda regla, ya que solo se ejecuta al iniciar la aplicación) utilizamos el ultimo mecanismo. Android proporciona un sistema llamado

ContentObserver que nos notifica de cambios en URIs del sistema, y podemos utilizarlo para escuchar cambios en la agenda. Cuando se detecta uno de estos cambios se buscan nuevos emails y teléfonos como en el segundo caso.

Si estas cuatro reglas no fallan, los usuarios deberían estar constantemente sincronizados y cada nuevo usuario aparecerá automáticamente. Aun así, el usuario tiene la opción de "Buscar cambios en los contactos" desde el menú contextual por si se produjese algún error.

3.2.5.2 CO-14: EDICIÓN DE NÚMEROS DE TELÉFONO

La aplicación usa los números de teléfono de los usuarios para localizar e identificar usuarios. Por ello es necesario que cada uno tenga un único número de teléfono asociado. Para poder usar este número como criterio de búsqueda posteriormente necesitamos que siempre se utilice el mismo formato de número.

En la agenda Android podemos encontrar números con el prefijo internacional (con o sin +, con o sin 00) o sin él, y con o sin espacios o guiones. Necesitamos unificar criterios, por ello todos los numero se convierten al formato internacional, sin los prefijos + ni 00, y sin espacios ni guiones. Para realizar estas conversiones entendemos que los números en la agenda que no tienen el código internacional pertenecen al mismo país que el usuario. Por ello añadimos el prefijo internacional a los números de la agenda cuando buscamos amigos usuarios, basándonos en el Locale de Android del usuario. Este proceso se realiza en la clase `ConversorPaisCodigo` y la lista de códigos numéricos y alfanuméricos empleados en la conversión se encuentra en res/values/códigos.xml.

3.2.5.3 OTROS PROCESOS AUTOMÁTICOS

En esta categoría podemos incluir el proceso que gestiona el GPS, descrito en Servicio y Posicionamiento y también algunos procesos que se producen cuando falla la comunicación y se reintenta, descrito en Comunicaciones.

3.2.6 CO-15: REPRESENTACIÓN EN EL MAPA

Este es un componente esencial cuya función es convertir las coordenadas geográficas a las coordenadas de los mapas que utiliza la aplicación.

Debe ser flexible y poder adaptarse a nuevos mapas. Este componente debe ser rápido, eficiente y realizar la conversión de coordenadas de manera precisa, utilizando el menor número de recursos posible.

3.2.7 INTERFAZ

La interfaz se basa en un patrón de diseño habitualmente utilizado en interfaces móviles. Se trata de un *layout* que utiliza pestañas en la parte superior para facilitar la navegación. Esta configuración nos permite elegir que pestañas contenedoras de actividades nos interesa tener en la aplicación. Habitualmente, querremos tener las pestañas relativas a la lista de amigos y al mapa. El uso de esta disposición nos permite añadir nuevas pestañas en función del contexto en que vayamos a utilizar la aplicación. Si por ejemplo el entorno elegido es el una estación de esquí, puede que nos interese tener una pestaña que nos dé información sobre el estado de la nieve y las pistas.

En la parte superior derecha hay un botón cuya función es activar o desactivar el servicio de posicionamiento. Esta función no depende de la actividad que este activa sino de la aplicación, por lo que el botón esta visible en todo momento. En modo retrato el botón se encuentra a la derecha de las pestañas y tiene el mismo tamaño.

Cuando ponemos la aplicación en modo apaisado, las pestañas pasan de estar en la parte superior al lateral izquierdo. El objetivo de este cambio es evitar que las pestañas ocupen demasiado espacio en este modo, restándoselo a las actividades. El grosor de las pestañas se reduce drásticamente, por lo que se elimina el indicador textual y se deja solamente el icono. El botón de posicionamiento, debido a estos cambios, pasa de estar integrado con las pestañas a la parte superior derecha. En este caso se superpone a la actividad seleccionada en ese momento por lo que hay que tenerlo en cuenta cuando se diseñan las actividades.

Para implementar todas las funciones deseadas, necesitamos crear una serie de vistas específicas, que describimos en las siguientes secciones.

FIGURA 3.7 - INTERFAZ

3.2.7.1 CO-16: MAPAVISTA

El objetivo de esta vista es mostrar por pantalla uno de los mapas que utiliza la aplicación e incluir una forma sencilla de dibujar vistas sobre el mismo. Por lo tanto se compone por un lado de la vista que representa el mapa y por otro de las vistas que se dibujan encima, en nuestro caso un indicador de nuestra posición e indicadores para las posiciones de los contactos.

Para el mapa necesitamos una vista que nos permita hacer zoom sobre la imagen con dos dedos. Android no proporciona ninguna vista que cumpla estos requisitos pero han sido desarrollados por terceros. En nuestro caso utilizamos la vista `ImageTouchView` [2]. Nos permite mostrar una imagen, el mapa, y movernos por la misma mediante gestos, ampliando, reduciendo y desplazando la misma.

3.2.7.2 CO-17: THEMISSINGTABHOST

Esta aplicación está pensada para utilizar un sistema de pestañas como forma de navegación. Este sistema funciona correctamente con el teléfono en posición vertical pero en modo

apaisado, las pestañas en la parte superior ocupan demasiado, dejando poco espacio al contenido. Esto no sería grave en otra aplicación, pero en ésta, especialmente cuando el usuario se encuentra en la actividad `MapaActivity`, se desaprovecha demasiado espacio en pantalla.

La solución pasa por mover las pestañas de la parte superior al lado izquierdo de la pantalla. El problema es que Android no proporciona una vista para implementar pestañas de esta manera por lo que tenemos que recurrir a soluciones de terceros. Encontramos `TheMissingTabHost` [3] que proporciona precisamente esta funcionalidad. Además lo hemos editado para eliminar el texto de las pestañas y reducir drásticamente el tamaño de las mismas en modo apaisado, para intentar maximizar el espacio.

3.2.8 COMUNICACIONES

En este apartado vamos a centrarnos en la interconexión de las dos grandes partes, el cliente Android y el backend en App Engine, o entre el modelo y la vista y el controlador, y los mensajes que se transmiten. En el cliente Android las comunicaciones las gestiona la clase Util.Comm, mientras que al otro lado se encuentran los servicios explicados en la sección Google App Engine.

3.2.8.1 CO-18: REGISTRO Y ENVIACONTACTOS

El usuario envía una petición de registro. Posteriormente envía sus datos (email y teléfono). En caso de recibir una respuesta satisfactoria inicia la búsqueda de amigos registrados, enviando todos los números de teléfono y emails encontrados en la agenda. Finalmente el servidor responde con una lista de contactos usuarios localizados.

3.2.8.2 CO-19: ENVIARPOSICION

Como su nombre indica, permite a un usuario enviar su posición. En caso de éxito, el servidor registra la posición del usuario y finaliza el proceso. En caso negativo, el cliente intentará reenviar la posición de nuevo con Exponential Backoff, (siempre la última posición actualizada).

3.2.8.3 CO-20: CONSULTAPOSICIONAMIGOS

Este método sirve para consultar las posiciones de los amigos. Cuando el servidor recibe esta consulta, hace una revisión de permisos y comprueba si el usuario puede acceder a las posiciones que solicita.

3.2.8.4 CO-21: CONSULTAPERMITIDOS Y ESTABLECEPERMISO

La función de este componente es que exista sincronía entre el estado de permisos de un usuario en el cliente y en el servidor.

3.2.8.5 CO-22: HACERPING Y POSICIONDISPONIBLE

Estos dos métodos son los encargados de implementar la funcionalidad Ping. La finalidad es permitir a un usuario consultar la posición de otro bajo demanda. Es posible que un contacto

no haya activado el servicio de posicionamiento o por problemas con la conexión de internet este tardando mucho en enviar su posición. En este caso un usuario puede solicitar a otro que envíe su posición.

3.3 GOOGLE APP ENGINE

En esta sección vamos a diseñar los componentes del modelo que se encargan de gestionar la parte de Modelo de nuestra aplicación

3.3.1 ENTIDADES

Las entidades representan objetos con capacidad de persistencia en el almacén de datos. Cada entidad contiene una clave que sirve de identificador para diferenciarla de las demás entidades. A su vez, cada entidad contiene una o varias propiedades, que se representan como atributos del objeto.

Las claves están compuestas por varios elementos: una ruta, un elemento opcional que determina la entidad principal de la entidad en cuestión, el tipo de entidad, así como el nombre asignado a la entidad por la aplicación o el ID numérico asignado por el almacén de datos. Cada entidad incluye una o varias propiedades, es decir, valores de uno de los distintos tipos de datos.

Vamos a ver los diferentes tipos de entidades que utiliza la aplicación:

3.3.1.1 CO-23: USUARIO

Representa a un usuario de la aplicación. La información que guardamos de cada uno tiene como único fin ayudar en la localización de amigos e identificación.

Tiene un campo ID, de tipo Key. La clase Key [6] está incluida en Google AppEngine y facilita la localización de entidades. En este caso, usamos el email del usuario para identificar este tipo de entidades. Así conseguimos facilitar su posterior recuperación.

También tienen un campo teléfono, cuyo objetivo es del mismo modo ayudar a la localización de los contactos que otro usuario tenga en su agenda.

3.3.1.2 CO-24: AMIGOS

Esta entidad representa una lista de contactos que el usuario conoce, es decir son usuarios de la aplicación y han sido localizados bien por mail o bien por el número de teléfono. No significa que el usuario pueda conocer la ubicación de los mismos, simplemente que están en su agenda y utilizan la aplicación. Se explica con más detalle en la sección Servicios.

Tiene un campo ID del mismo tipo que la entidad Usuario, utiliza el email del usuario como clave, y una lista de usuarios que representa lo descrito.

3.3.1.3 CO-25: PERMITIDOS

Esta entidad también engloba una lista de usuarios, pero este caso incluye los contactos a los que el usuario definido por la clave ha concedido permisos. Esta entidad debe estar siempre sincronizada con el terminal del usuario.

Tiene un campo ID del mismo tipo que la entidad Usuario, utiliza el email del usuario como clave, y una lista de usuarios que representa lo descrito.

3.3.1.4 CO-26: POSICIÓN

Representa la posición de un usuario a través de las coordenadas longitud y latitud.

Tiene un campo ID del mismo tipo que la entidad Usuario, utiliza el email del usuario como clave, y un objeto Location.

3.3.1.5 CO-27: CONTACTOS

La función de esta entidad es mantener una copia de la agenda telefónica del usuario para facilitar la localización de nuevos amigos, después de la primera ejecución donde se buscan contactos.

Consta de una lista de teléfonos y otra lista de emails. En ningún caso se relaciona ninguno de estos datos de ninguna manera, ni con nombres de usuario, ni entre los propios teléfonos o emails. No es posible saber que email pertenece a que usuario ni vincularlo con su teléfono. Estos datos se utilizan única y exclusivamente para sugerir nuevos amigos a los usuarios.

3.3.2 SERVICIOS

La aplicación necesita de una serie de servicios para para poder acceder y modificar las entidades descritas anteriormente. A pesar de existir múltiples alternativas, en nuestro caso hemos elegido adoptar un diseño orientado a datos.

Hemos separado los servicios en dos categorías, uno encargado de la gestión de usuarios y otro encargado de la gestión de las posiciones.

3.3.2.1 CO-28: REGISTROREQUEST

Se trata de un componente esencial en el diseño de la aplicación al igual que el siguiente CO-29. Se trata de una interfaz que permitirá hacer llamadas remotas desde el cliente utilizando objetos de tipo Proxy, que son reflejos de las entidades del servidor.

Este componente debe permitir al cliente registrar usuarios, gestionar contactos y gestionar permisos.

3.3.2.2 CO-29: POSICIONREQUEST

Del mismo modo que el anterior representa una interfaz común que comparten cliente y servidor y que es invocada a través de RequestFactory. En este caso se encarga gestionar las posiciones de los usuarios, el registro y las consultas de las mismas, además de implementar la parte del servidor de la función Ping.

3.3.3 DATASTORE

Hasta ahora tenemos entidades y servicios que operan sobre ellas, pero necesitamos almacenar esas entidades de alguna forma. El sistema que vamos a utilizar para garantizar la persistencia de objetos es JDO.

3 DISEÑO

Persistir objetos Java en una base de datos relacional es un reto único que implica serializar un árbol de objetos Java en una base de datos de estructura tabular y viceversa. Este reto actualmente está siendo corregido por varias herramientas diferentes. La mayoría de estas herramientas permiten a los desarrolladores instruir a los motores de persistencia en cómo convertir objetos Java a columnas/registros de la base de datos y al revés. Es esencial para este fin la necesidad de mapear objetos Java a columnas y registros de la base de datos de una forma optimizada.

El API Java Data Objects (JDO) proporciona una forma estándar y sencilla de conseguir la persistencia de objetos en Java. JDO utiliza una combinación práctica de metadatos XML y bytecodes mejorados para hacer más sencilla la complejidad y la sobrecarga, comparado con otras tecnologías de unión de objetos.

El API JDO, consta sólo de unos pocos interfaces, es uno de los APIs más fáciles de aprender de toda la tecnología existente actualmente de persistencia. Hay muchas implementaciones de JDO entre las que elegir. El motor de Google App Engine incluye una, por lo que no tenemos que preocuparnos por la instalación.

Una razón de la simplicidad de JDO es que permite trabajar con objetos normales de Java (POJOs plain old Java objects) en lugar de con APIs propietarios. Proporciona una capa de abstracción entre el código de la aplicación y el motor de persistencia.

En la aplicación el acceso al datastore se realiza directamente desde los servicios. Podría parecer interesante crear una clase separada que se encargase de tal tarea, pero el funcionamiento de JDO es tan sencillo que no hace falta.

3.4 MATRIZ DE TRAZABILIDAD

En esta matriz, establecemos la relación entre requisitos y componentes.

	CO-01	CO-02	CO-03	CO-04	CO-05	CO-06	CO-07	CO-08	CO-09	CO-10	CO-11	CO-12	CO-13	CO-14	CO-15	CO-16	CO-17	CO-18	CO-19	CO-20	CO-21	CO-22	CO-23	CO-24	CO-25	CO-26	CO-27	CO-28	CO-29
UR-01		X																											
UR-02		X																											
UR-03										X			X																
UR-04				X																									
UR-05							X														X								
UR-06					X										X	X													
UR-07																X													
UR-08	X											X																	
UR-09																						X							
UR-10	X	X	X	X	X	X	X	X																					
UR-11				X														X											
UR-12												X						X											
UR-13																		X	X	X	X	X	X	X	X	X	X	X	X
UR-14						X	X																						
UR-15			X								X								X	X								
UR-16					X										X	X													
UR-17	X														X	X													
UR-18							X														X								
UR-19															X														

4 IMPLEMENTACIÓN

En esta sección vamos a tomar como referencia los componentes diseñados en la sección anterior e indicar como se han resuelto problemas surgidos en el desarrollo, o detalles no resueltos en la misma.

En Android se separa claramente el diseño del resto del código, al estilo de la programación web. El diseño aquí se establece mediante unos archivos XML donde se describen los Widgets. Cada actividad vincula su *layout* a un archivo XML, de forma similar a como una página HTML se vincula a un fichero CSS.

4.1 ACTIVIDADES

4.1.1 CO-01: TABLAYOUTACTIVITY

Esta clase tiene tres funciones principales.

La primera y más obvia es la de establecer el tipo de *layout* como un sistema de pestañas. Basándonos en el componente CO-17, establecemos el número de pestañas, los nombres e iconos de las mismas.

Esta actividad es la actividad principal, la primera que se inicia y por lo tanto tiene que ocuparse de activar la actividad de registro si es necesario. Al abrir la aplicación se comprueba que el usuario se ha registrado. En caso negativo, abre automáticamente la actividad de registro. La actividad de registro puede cerrarse con éxito o con fallo. Si se termina con éxito,

esta actividad se encarga de pedir el número de teléfono del usuario y de iniciar la búsqueda de contactos en la agenda. Si termina en fracaso significa que el usuario ha salido de la actividad de registro pulsando el botón de atrás. En ese caso se cierra la aplicación. El usuario no percibe nada extraño, porque nunca ha visto esta actividad, sino que directamente ha visto el registro. Al pulsar atrás está pidiendo cerrar la única actividad que ha visto. En caso de que el usuario ya estuviese registrado, la aplicación hace login con el servidor.

La última función es gestionar el botón que controla el servicio de posicionamiento. Al iniciar la actividad se comprueba en qué estado está el servicio, estableciendo el botón como activado o desactivado. También captura las pulsaciones en el mismo y transmite las instrucciones.

4.1.2 CO-02: CUENTASACTIVITY

La función de esta actividad es conectar la aplicación con los servidores de Google. Esta actividad tiene dos modos, conectado y desconectado. Tiene dos archivos de diseño XML diferentes para cada caso y el código contempla ambas opciones. En caso de estar conectado la actividad accede a una lista de cuentas Google utilizando un objeto `AccountManager`. El usuario puede elegir una de las cuentas y pulsar en Conectar. La actividad intenta conectar con el servidor de Google. En caso de éxito la actividad finaliza con resultado positivo. En caso contrario muestra un mensaje de error y no finaliza. Si el usuario pulsa el botón Atrás, finaliza con resultado negativo.

4.1.3 CO-03: POSICIONLISTENERACTIVITY

Como decíamos en la sección de Diseño, esta actividad abstracta se encarga de recibir las posiciones propias y de los contactos. Las posiciones propias se reciben conectándose al servicio de posicionamiento (CO-12) y suscribiendo un BroadcastReceiver para escuchar las mismas. Las actualizaciones de los contactos se consultan periódicamente al servidor. Ambas actualizaciones son transmitidas a las clases herederas mediante métodos abstractos y se deja que ellas se encarguen de hacer con ellas lo que consideren.

Los métodos abstractos son los siguientes:

```
/**
 * Se llama cuando el servicio devuelve una nueva posición
 */
protected abstract void onPosicionActualizada(Location location);

/**
 * Se llama cuando se recibe la posición de los amigos
 */
protected abstract void onPosicionAmigosActualizadas(List<Amigo> amigos);
```

onPosicionActualizada se llama cuando la posición del usuario varía y onPosicionAmigosActualizadas es llamado cuando se reciben nuevas posiciones de los contactos. De esta manera la clase heredera puede decidir qué hacer con los parámetros recibidos.

onPosicionActualizada se llama a través de un BroadCastReceiver que recibe un Intent enviado por el servicio con la nueva posición. onPosicionAmigosActualizadas se llama del mismo modo, pero cuando se detecta un cambio en la base de datos de contactos; así cuando hay un cambio se lee la nueva situación de la base de datos y se transmite por este método abstracto.

4.1.4 CO-04: LISTAAMIGOSACTIVITY

Esta actividad tiene que mostrar la lista de amigos usuarios. Como hemos dicho, hereda de la actividad PosicionListenerActivity que hace buena parte de su trabajo. En realidad se encarga de tomar la lista de contactos recibidos y pasárselos a un ListAdapter. Cada elemento de la lista es un contacto. Se establece el nombre, la foto de contacto, el nombre de la zona dentro del mapa y el tiempo pasado desde la última actualización.

En el menú contextual se da la opción de buscar nuevos contactos, establecer permisos, abrir la actividad de las cuentas o abrir la actividad de configuración.

También se incluye un gran botón en la parte inferior que lleva al usuario a la actividad de permisos.

4.1.5 CO-05: MAPAACTIVITY

Como en el caso anterior, esta clase también hereda de PosicionListenerActivity por lo que escucha los cambios en la posición propia y la de los contactos. Esta actividad detecta automáticamente si el usuario se encuentra en algún recinto acotado por un mapa y de ser así lo abre automáticamente.

El mapa trabaja conjuntamente con el componente MapaVista (CO-16). La funcionalidad se divide de la siguiente forma. La actividad implementa un archivo de diseño XML que contiene un Widget de tipo MapaVista. Además de esta vista, está encargado de representar sobre ella la posición del usuario y de los contactos. El componente de la vista se encarga de capturar los eventos del usuario, desplazamientos, zoom y pulsaciones. El componente modifica el tamaño y posición del mapa y transmite ese cambio a la actividad.

Este cambio se notifica a la actividad pasándole un rectángulo. Este rectángulo representa la posición relativa del mapa respecto a la pantalla del dispositivo. El rectángulo tiene 4 coordenadas:

- X inicial
- X final
- Y inicial
- Y final

Los valores pueden ser positivos o negativos. La actividad conoce el tamaño de pantalla asignado al mapa y mediante estos valores puede saber si el mapa es más grande que la pantalla o está incluido entero. En base a esa información representa la posición propia y de los contactos en las posiciones absolutas de la pantalla (o no si no se encuentran en la zona del mapa visible).

Para cada vista que representa la posición de un contacto en el mapa mediante la foto de contacto o mediante una foto genérica si no está establecida y se establece un Listener de

eventos que captura las pulsaciones del usuario sobre las mismas. Cuando un usuario pulsa sobre ella, se muestra un label de información en la parte superior donde se indica el nombre del usuario según figura en la agenda y la última vez que se le vio.

Dependiendo de en qué contexto se utilice la aplicación es posible que el número de mapas sea demasiado elevado para incluir todos en el paquete de instalación. De ser así, los mapas se descargarán bajo demanda. Por ejemplo en el entorno de las estaciones de esquí, cuando seleccionamos una la primera vez, se descargará el mapa una única vez y se quedará guardado en la tarjeta de memoria para usarlo en el futuro.

4.1.6 CO-06: VISTAAMIGOACTIVITY

Esta actividad se abre siempre como un dialogo flotante sobre la actividad que la llama, siempre una de las dos anteriores. El desarrollo de la misma es sencillo. Se carga el *layout* desde el archivo XML correspondiente, donde se establece su carácter de dialogo flotante. Después se establecen los parámetros nombre, zona del mapa en que se encuentra y última vez que se le vio. El botón Ping activa la funcionalidad descrita en el apartado de Diseño.

4.1.7 CO-07: PERMISOSAMIGOSACTIVITY

Esta actividad se encarga de mostrar una lista con los contactos que usan la aplicación y permitir asignar o revocar el permiso de ver la ubicación propia a cada uno de ellos.

La implementación se basa en crear una lista de los contactos conocidos, que se extrae de la base de datos de la aplicación. Para cada elemento se establece el nombre, la foto y un botón que puede tener dos estados. Los estados son dos iconos que representan si se está concediendo permiso al usuario indicado para conocer la ubicación propia o no. Al hacer click en ese botón, se desactiva temporalmente y se inicia el proceso de cambio de estado. Este proceso consiste en pedir al servidor un cambio de estado. Si se produce con éxito el botón se activa de nuevo con el icono que representa el nuevo estado. Si se produce un fallo, el botón se activa también, pero el icono representa el estado anterior y se muestra un mensaje indicando que ha habido un fallo en la comunicación.

El estado inicial de los permisos se obtiene llamando de forma asíncrona al método que se encarga de conocer el estado de los permisos en el servidor (CO-21).

El funcionamiento es sencillo, cada elemento de la lista tiene el nombre y la foto del usuario, y un botón que puede tener dos estados, dependiendo de si el permiso esta concedido o no. En caso de pulsar el botón, se desactiva y el icono cambia a "cargando". Tras comunicarse con el servidor, el botón adquiere el nuevo estado, teniendo en cuenta si se ha establecido con éxito o no.

4.1.8 CO-08: PREFERENCIASACTIVITY

Esta actividad sigue el patrón estándar de menú de preferencias de Android y tiene únicamente una opción de selección, la de activar el posicionamiento de forma automática al iniciar la aplicación.

El código es tan simple que podemos incluirlo por entero aquí:

```
public class PreferenciasActivity extends PreferenceActivity {

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		requestWindowFeature(Window.FEATURE_NO_TITLE);
		super.onCreate(savedInstanceState);

		// Carga las preferencias de un archive XML
		addPreferencesFromResource(R.xml.preferences);

	}

}
```

Simplemente hereda de PreferenceActivity de Android y establece el archivo XML en que esta el diseño. Ese archivo representa únicamente una opción que permite al usuario indicar si quiere que el posicionamiento se active de forma automática al iniciar la aplicación o no.

4.1.9 DIAGRAMA

A continuación se incluye un diagrama de clases del paquete, donde se incluyen los métodos de cada una.

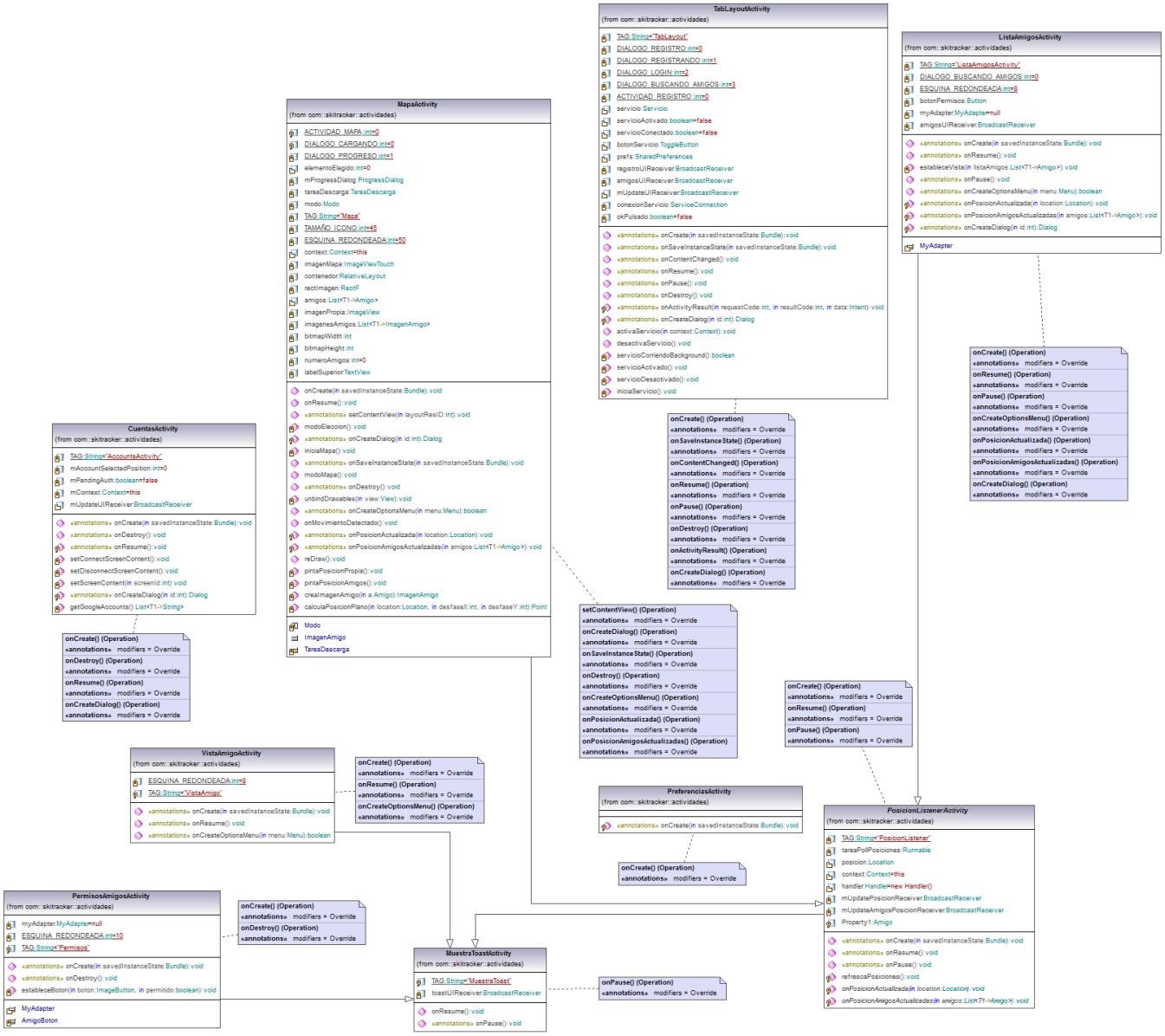

FIGURA 4.1 - DIAGRAMA DE ACTIVIDADES

4.2 ALMACENAMIENTO

4.2.1 CO-09: SQLITE

La clase `Amigo` es una clase envolvente que maneja la aplicación y representa un contacto que utiliza la aplicación. Tiene los siguientes atributos y los correspondientes getters and setters.

```
private int ID;
private String nombre;
private String telefono;
private String email;
private Location posicion;
private boolean permitido;
```

El ID coincide con el ID que emplea Android para identificar a los contactos en la agenda. Nos sirve para recuperar la foto y el nombre. El atributo permitido indica si estamos compartiendo nuestra ubicación con este usuario.

Nuestro gestor de base de datos se encarga de guardar y recuperar este tipo de entradas del almacenamiento del teléfono. Incluye métodos para realizar las siguientes funciones:

- Crear la base de datos
- Guardar una lista de amigos usuarios recibida del servidor
- Guardar una lista de los permisos que tenemos asignados a cada usuario del servidor
- Guardar una lista de posiciones de usuarios recibida del servidor
- Devolver la lista de objetos amigo del tipo anterior
- Devolver un único amigo a partir de su ID
- Establecer el permiso para un único amigo a partir de su email
- Borrar la base de datos

Como hemos dicho esta clase asegura la persistencia de datos entre actualizaciones desde el servidor. De no utilizarse, cada vez que iniciásemos una actividad, tendríamos que pedir los datos de los contactos, produciendo esperas y retransmisiones innecesarias.

4.2.2 CO-10: ALMACENAMIENTO EXTERNO

La clase `Contactos` de tipo serializable se encarga de almacenar una lista de teléfonos y emails que ya han sido consultados con el servidor para evitar que cada vez que iniciamos la aplicación no se busquen una y otra vez nuevos usuarios, sino solo sobre los nuevos contactos en la agenda. Simplificando, mantiene una copia de la agenda en el momento en que se contrastó la última vez con el servidor, para evitar volver a contrastar determinados contactos.

Esta clase incluye métodos estáticos para recuperar la lista de teléfonos y emails tal y como está en la agenda en el momento actual, recuperar la misma lista según se guardó la última vez y guardar una nueva lista.

El funcionamiento se reduce a serializar el objeto contactos y guardarlo y leerlo utilizando el sistema de acceso a ficheros de Android.

4.2.2.1 DIAGRAMA

A continuación se incluye un diagrama de clases del paquete, donde se incluyen los métodos de cada una.

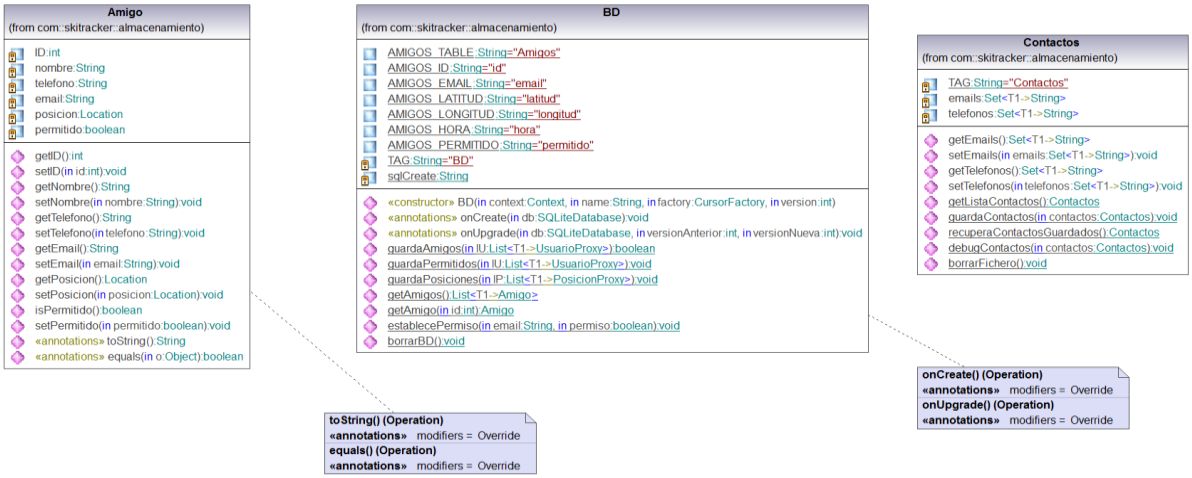

FIGURA 4.2 - DIAGRAMA DE ALMACENAMIENTO

4.3 SERVICIO Y POSICIONAMIENTO

4.3.1 CO-11: POSICIONAMIENTO

En esta clase se definen dos objetos de tipo Runnable que se encargan uno de activar el Posicionamiento, llamando al LocationManager y el otro de parar el posicionamiento en caso de timeout.

El resto de la clase proporciona métodos públicos para poder ejecutar estos Threads, pararlos u obtener las posiciones recuperadas. El funcionamiento de esta clase consiste en iniciar el posicionamiento llamando al LocationManager y solicitando nuevas posiciones de los proveedores de GPS y basados en redes WIFI.

Estas posiciones se capturan extendiendo el método abstracto:

```
public void onLocationChanged(Location location)
```

Al recibir una nueva posición, la comparamos con la última recibida. Solo continuamos el proceso si ha pasado un determinado tiempo desde la última posición o si la precisión es mejor. Si la posición nos resulta interesante, paramos el posicionamiento para ahorrar energía y programamos el inicio para un minuto más tarde. El tiempo es configurable.

El siguiente paso es comprobar si ha pasado suficiente tiempo desde la última vez que enviamos la posición al servidor. Si es así, se envía la posición. En caso de error se reintenta, a no ser que haya una posición nueva, en cuyo caso se envía la más actualizada.

Cuando se recibe una nueva posición valida, también nos encargamos de meterlo en un Intent y hacer Broadcast para que puedan recibirlo las clases que heredan de PosicionListenerActivity.

Por ultimo incluye un modo de posicionamiento en que se espera una única posición antes del timeout, y se desactiva el posicionamiento. Este modo se usa cuando se recibe un Ping y el posicionamiento no está activado.

4.3.2 CO-12: SERVICIO

La implementación del servicio resulta bastante sencilla en la práctica ya que su funcionalidad principal, el posicionamiento, esta delegada en el componente homónimo. El servicio hereda de la clase Service e incluye los métodos que hay que heredar de la misma.

El servicio tiene varios modos. Uno se ejecuta cuando el usuario no está utilizando la aplicación pero ha activado el servicio de posicionamiento. En ese caso se solicita una posición por minuto y se envía al servidor, desactivando el posicionamiento entre medias, como indicábamos en el componente anterior.

El segundo modo se ejecuta cuando el usuario si está utilizando la aplicación. En ese caso el posicionamiento está constantemente activado con la función de actualizar la UI con la mayor rapidez posible, por ejemplo si estamos en la actividad Mapa y nos hemos movido.

El servicio incluye un método onBind, que permite a actividades conectarse al mismo y obtener el objeto Servicio que se está ejecutando. De esta forma las actividades pueden cambiar el modo del servicio cuando consideren.

Por último el servicio también es el encargado de mostrar la notificación en la barra de tareas cuando el servicio de posicionamiento está activado. Si el usuario pulsa sobre la notificación, se abre la aplicación.

4 IMPLEMENTACIÓN

A continuación se incluye un diagrama de secuencia donde se detalla la interacción entre Servicio y Posicionamiento:

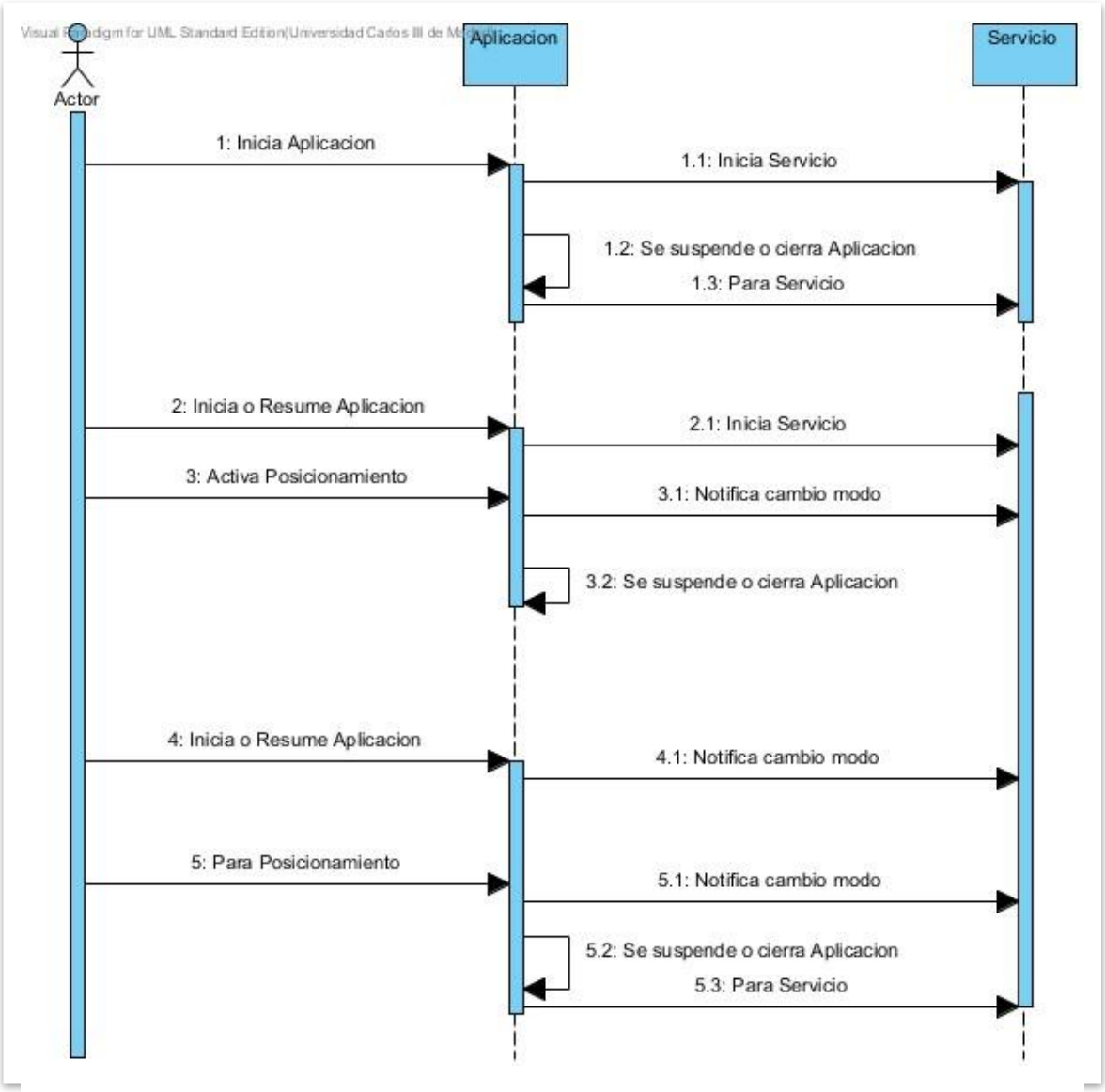

FIGURA 4.3 - CICLO DE VIDA DEL SERVICIO

4.4 PROCESOS AUTOMÁTICOS

4.4.1 CO-13: BÚSQUEDA AUTOMÁTICA DE CONTACTOS

Como decíamos en la sección de Diseño, el objetivo de este componente es que al iniciar la aplicación la lista de contactos se actualice espontáneamente sin que el usuario tenga que buscar en la agenda o introducir emails manualmente o conectarse a algún servicio externo como una red social. La implementación de este componente se apoya en buena medida en el componente CO-10, y la clase Contactos descrita en él.

Al iniciar la aplicación por primera vez se envían los teléfonos y emails encontrados en la agenda. Se crea un objeto Contactos con las listas y se guardan. En sucesivas ejecuciones solo se envían los nuevos emails y teléfonos encontrados en la agenda y no en el fichero Contactos almacenado. Así se evitan retransmisiones.

Por otro lado, puede pasar que un usuario empiece a utilizar la aplicación cuando ya lo hemos comprobado. Por ello se guarda en el servidor una lista de contactos de cada usuario, y se notifica al cliente en caso de encontrar nuevos usuarios entre los contactos de otro.

La implementación de este servicio se consigue utilizando mensajes push C2Dm, que indican al cliente que tiene nuevos contactos usuarios.

4.4.2 CO-14: EDICIÓN DE NÚMEROS DE TELÉFONO

Es necesario que todos los números se guarden en el mismo formato. El formato utilizado es un String del tipo:

+[prefijo país][numero teléfono]

Sin espacios, ni guiones.

Al pedir el número de teléfono al usuario se realizan varios pasos. Primero averiguamos el Locale del usuario, consiguiendo un código de identificación de país como ES o DE o IT. Este código lo buscamos en un fichero XML y lo convertimos al código numérico como 34, 44 o 49.

El código obtenido se lo sugerimos al usuario, permitiéndole cambiarlo y dejamos otro dialogo para que introduzca su número. Después se añade el + inicial y se juntan ambos campos formando un String estándar. Este proceso se realiza en la clase `ConversorPaisCodigo` y la lista de códigos numéricos y alfanuméricos empleados en la conversión se encuentra en res/values/códigos.xml.

4.5 CO-15: REPRESENTACIÓN EN EL MAPA

Este componente se encarga de representar al usuario y sus contactos sobre el mapa. Es necesario un sistema para convertir las coordenadas geográficas a coordenadas del plano que hemos elegido en la sección Diseño.

4.5.1 PRIMERA IMPLEMENTACIÓN

La solución adoptada en principio fue la siguiente:

- Se toma un determinado número de puntos significativos del plano que usaremos en la aplicación.
- Se buscan las coordenadas en longitud y latitud equivalentes a los puntos anteriores.

De esta forma obtenemos un mapeado de puntos del plano con sus coordenadas geográficas equivalentes.

El objetivo es que dada cualquier posición geográfica, podamos obtener el equivalente el pixeles en el plano. De momento solo tenemos una serie de puntos en un sistema de coordenadas continuo.

4 IMPLEMENTACIÓN

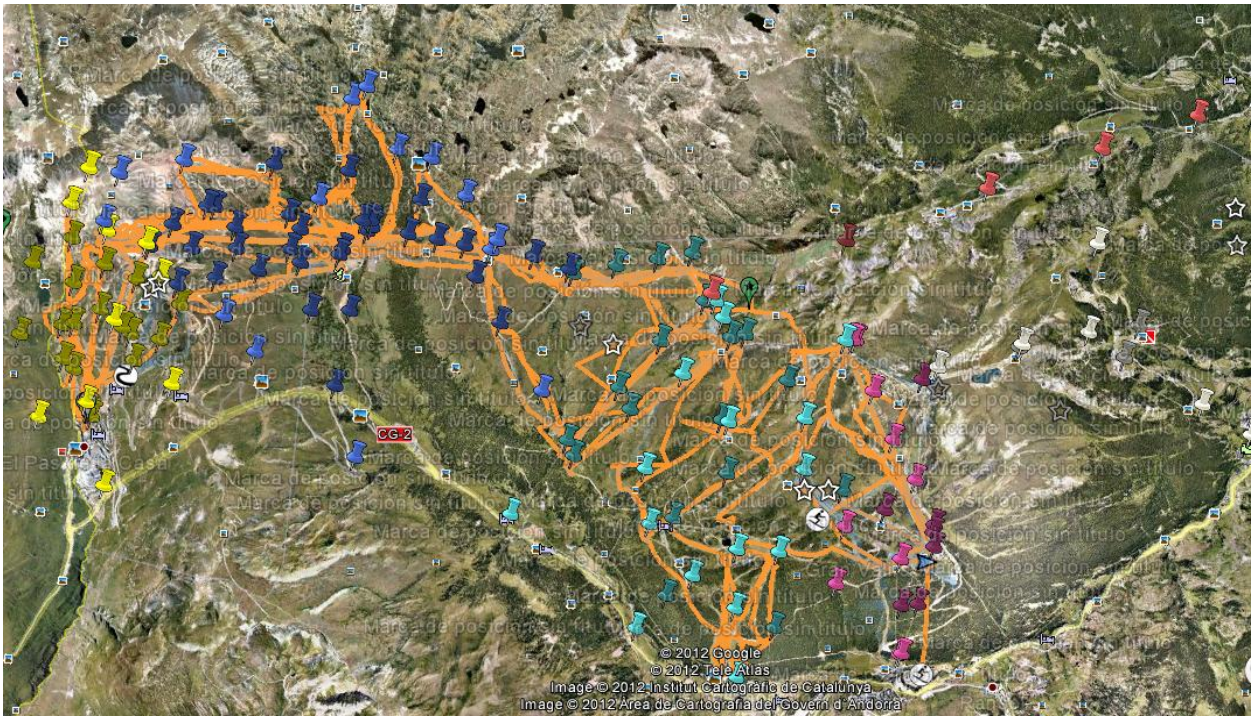

FIGURA 4.4 - COORDENADAS GEOGRÁFICAS SELECCIONADAS EN GRAND VALIRA

En esta imagen podemos ver una serie de 'pins' que representan los puntos que hemos tomado como referencia. Las líneas naranjas las hemos obtenido al superponer rutas que otros usuarios han grabado esquiando. De esta manera podemos interpretar esas líneas y buscar por donde circulan las pistas y los remontes. Estas rutas han sido descargadas en formato GPX desde Wikiloc [12].

En la siguiente ilustración buscamos los puntos equivalentes en el mapa.

4 IMPLEMENTACIÓN

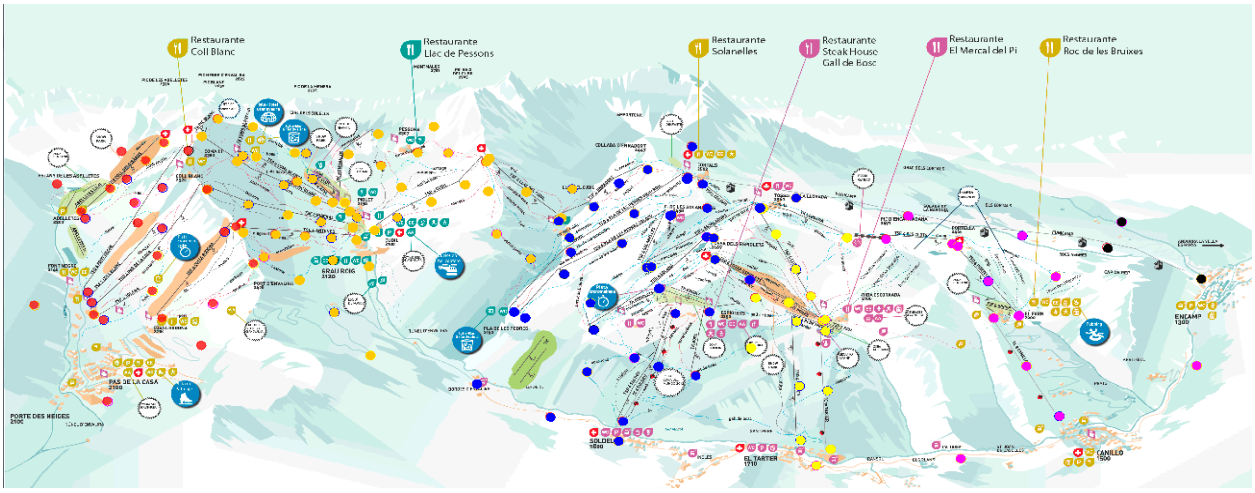

FIGURA 4.5 - COORDENADAS EQUIVALENTES EN EL PLANO DE GRAND VALIRA

Veamos un ejemplo del modo en que se mapean las coordenadas geográficas con puntos del mapa.

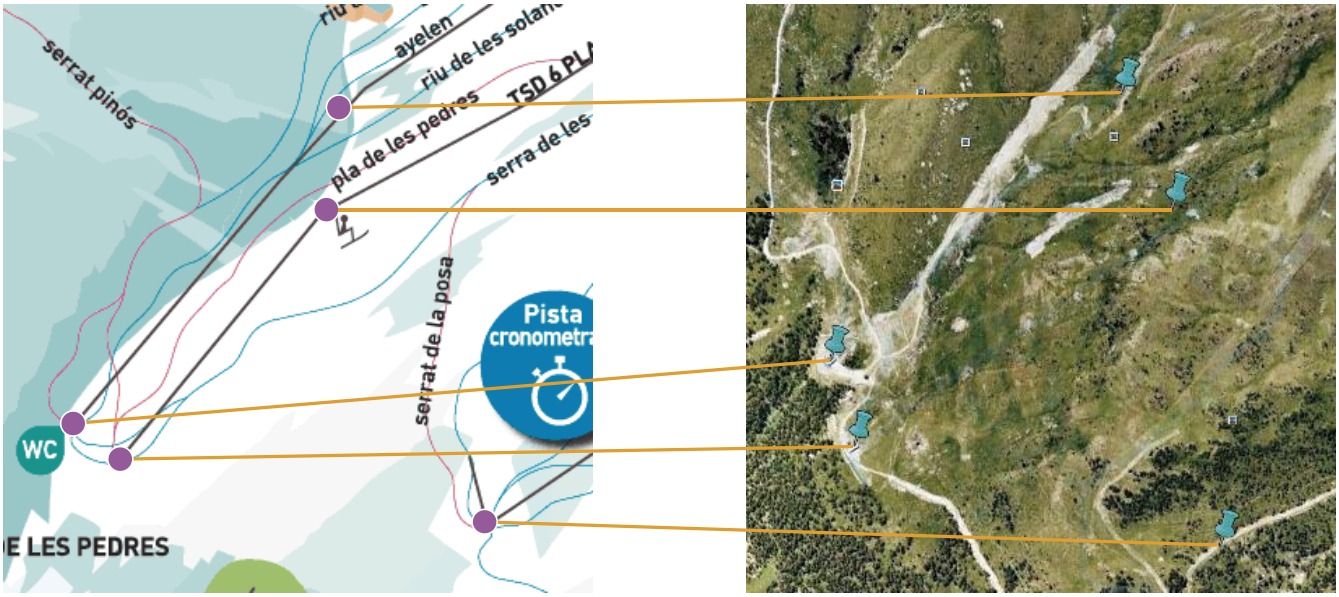

FIGURA 4.6 - RELACIÓN ENTRE PUNTOS DEL PLANO Y COORDENADAS GEOGRÁFICAS

Una vez establecidas las relaciones entre realidad y plano, necesitamos convertir los puntos en una malla de triángulos. La condición para que tres puntos formen un triángulo en nuestro mapa es: trazamos la circunferencia, con centro en el circuncentro, que pasa por los tres puntos y dicha circunferencia no contiene a ningún otro punto. De este modo garantizamos una malla de triángulos única y óptima.

Podemos ver un ejemplo en la Figura 4.7

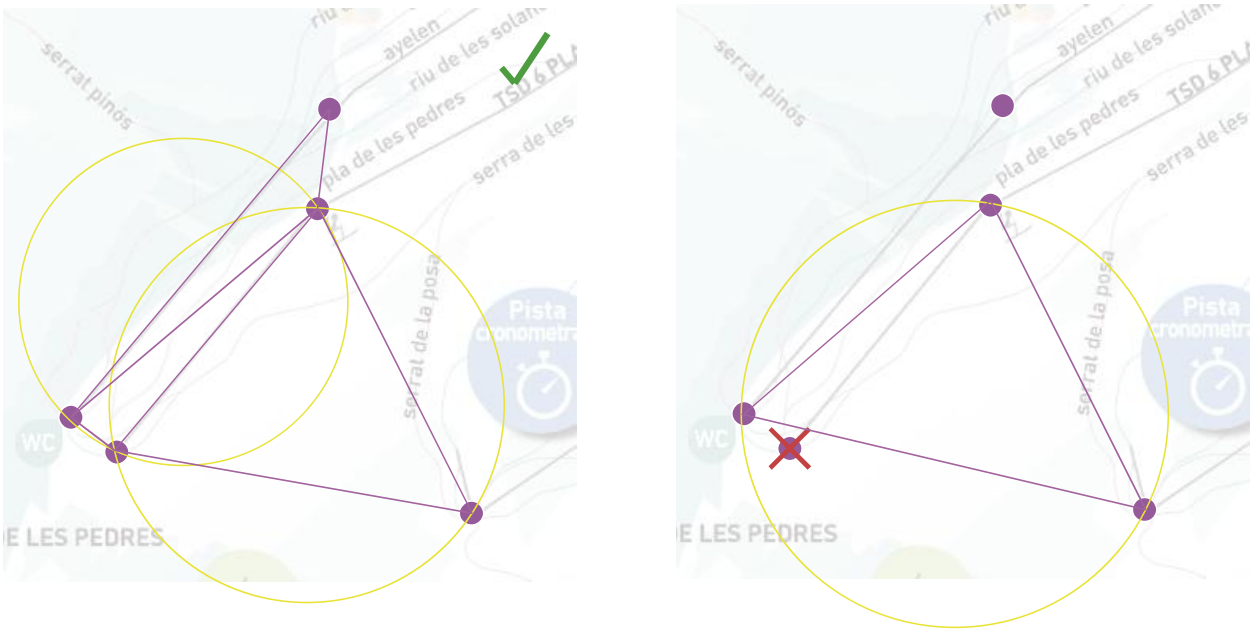

FIGURA 4.7 - TÉCNICA DE TRIANGULACIÓN

Como es natural, los triángulos formados en el plano son equivalentes a los localizados en la realidad por coordenadas geográficas.

Para convertir una coordenada geográfica a una coordenada (x, y) del plano:

- Buscamos el triángulo en el que se encuentra
- Calculamos el baricentro, dividiendo el triángulo en otros 3

- De esos tres triángulos, seleccionamos el que incluye al punto en coordenadas reales, que tiene un triángulo equivalente en coordenadas del plano.
- Repetimos el proceso en el nuevo triangulo hasta llegar a un triángulo lo suficientemente pequeño. En ese caso nos quedamos con el baricentro del mismo y damos la conversión por finalizada.

Se ilustra el proceso en la siguiente Figura 4.8.

FIGURA 4.8 - PROCESO DE CONVERSIÓN DE COORDENADAS (PARTE I)

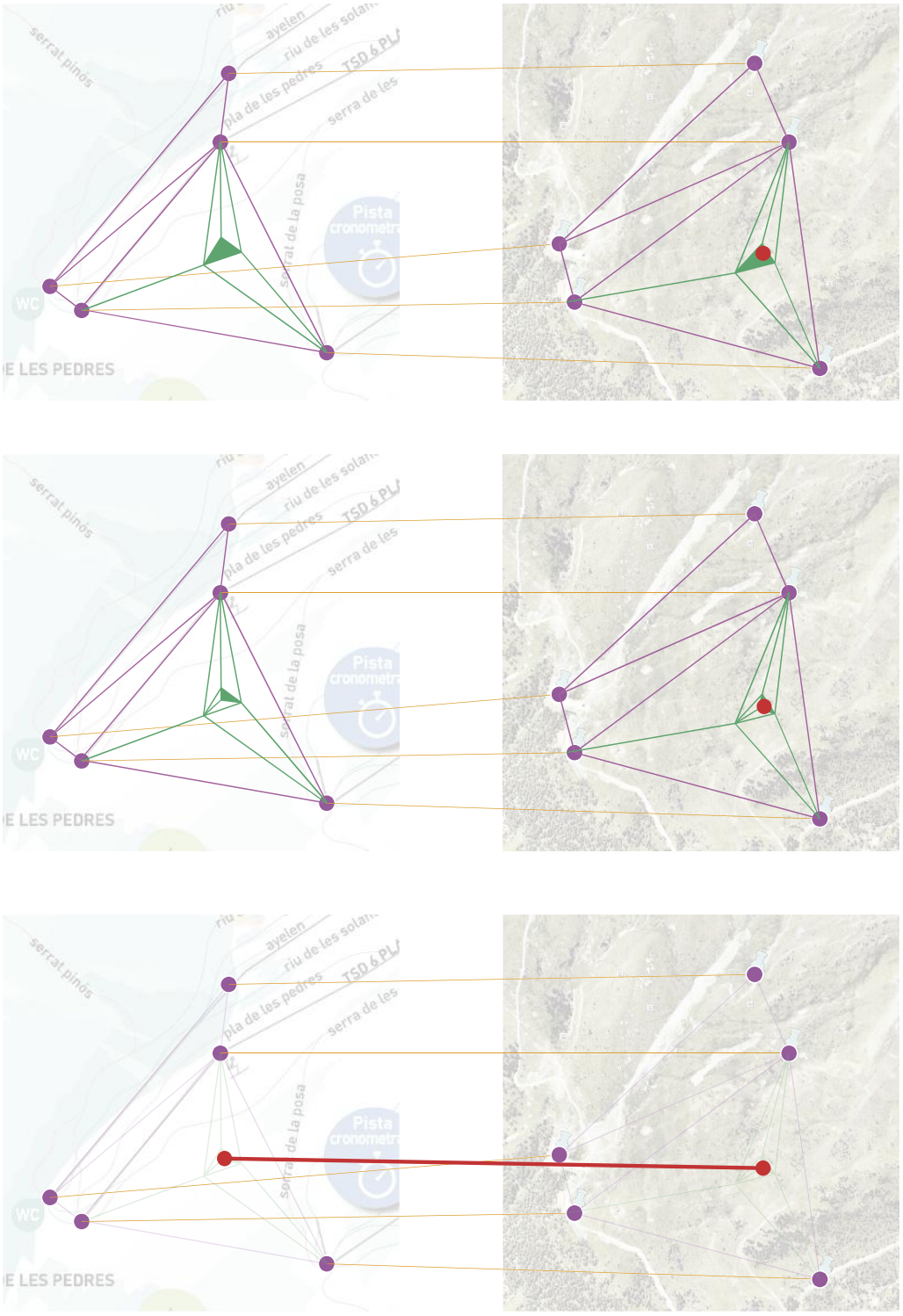

FIGURA 4.9 - PROCESO DE CONVERSIÓN DE COORDENADAS (PARTE II)

Como vemos el proceso se resume en localizar el triángulo en que se encuentra el punto original, dividirlo en tres y repetir el proceso. La complejidad de este método es de $O(n^3)$.

4.5.2 EVOLUCIÓN BASADA EN MINERÍA DE DATOS

Como hemos dicho, el sistema de conversión de coordenadas que hemos creado hasta ahora es costoso computacionalmente.

Los dispositivos móviles en los que pretendemos ejecutar este algoritmo tienen limitaciones principalmente en:

- Capacidad de procesamiento
- Memoria principal
- Duración de la batería

Por ello ejecutar un algoritmo de complejidad $O(n^3)$ sobre ellos, cuando es posible que haya cientos de puntos por mapa por evaluar, es una mala idea. Especialmente sabiendo que el Receptor del GPS puede devolver del orden de unas 10 posiciones por segundo, y que estas posiciones son las que tienen que ser interpretadas.

Por ello es necesario buscar una alternativa. Aunque los mapas representados no tienen correspondencia escalar directa de ningún tipo con las coordenadas geográficas, parece intuitivo pensar que podría existir algún sistema de reglas linear que convierta unas coordenadas en otras.

Existen varios algoritmos de minería de datos basados en reglas de asociación pero hay uno que cumple perfectamente nuestras necesidades: M5. Para la realización de estas pruebas emplearemos el proyecto de software libre Weka [9], de la universidad de Waikato.

4.5.3 INTERPOLACIÓN DE PUNTOS

Si ejecutamos el algoritmo M5 de reglas con los puntos que hemos seleccionado. Obtenemos una precisión relativamente buena (para los puntos conocidos) pero no es capaz de extrapolar e interpretar coordenadas que no se parecen a las conocidas.

Necesitamos más puntos de entrenamiento. La forma de hacerlo será creando una malla de triángulos como hemos descrito en la sección Conversión de Coordenadas. De este modo tenemos dos mallas de triángulos, uno en coordenadas geográficas y otro equivalente en coordenadas en pixeles sobre el plano. Por ello entendemos que entre dos de estos triángulos equivalentes, sus baricentros también serán equivalentes.

Por ello lo que hacemos es:

- Generar la malla de triángulos a partir de los puntos conocidos
- Calcular los baricentros
- Añadir los baricentros como nuevos puntos.

El formato utilizado es el siguiente:

```
42.540919, 1.732477, 303, 1089, "Pas de la Casa"
```

Los campos en orden representan:

- Latitud geográfica
- Longitud geográfica
- Coordenada x en el plano
- Coordenada y en el plano
- Nombre de la zona en que se encuentra

En este caso tras dos iteraciones obtenemos 24660 puntos.

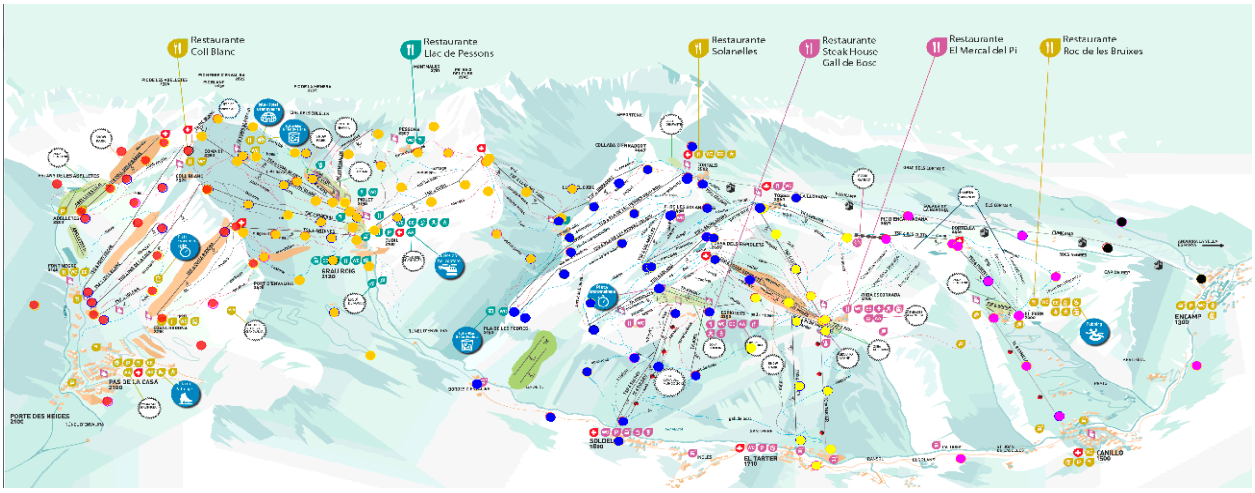

FIGURA 4.10 - ELECCIÓN MANUAL DE 152 PUNTOS DEL MAPA DE GRAND VALIRA

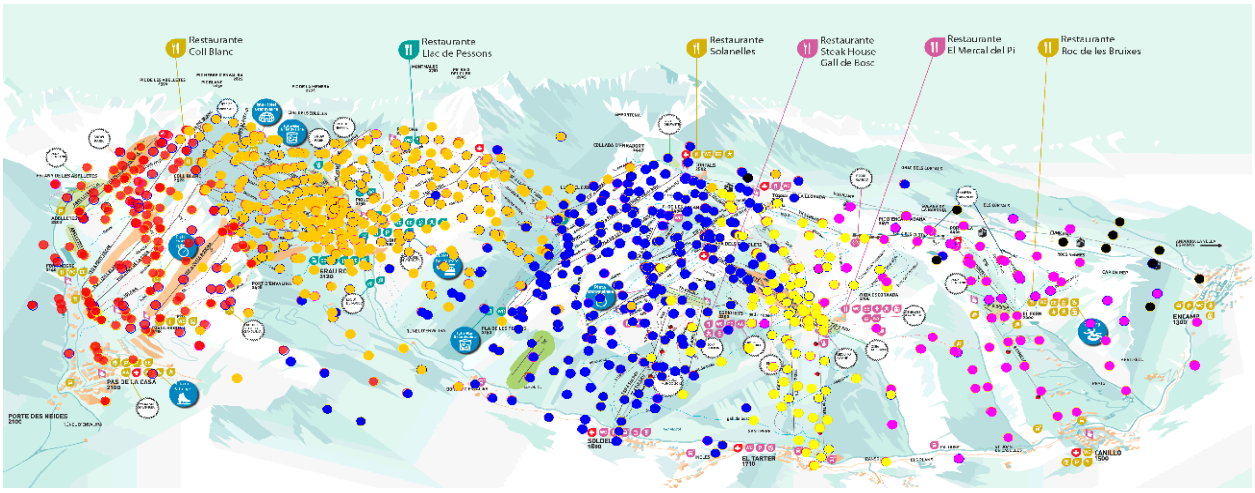

FIGURA 4.11 - PRIMERA EXTRAPOLACIÓN DE PUNTOS, CON UN TOTAL DE 1096 PUNTOS

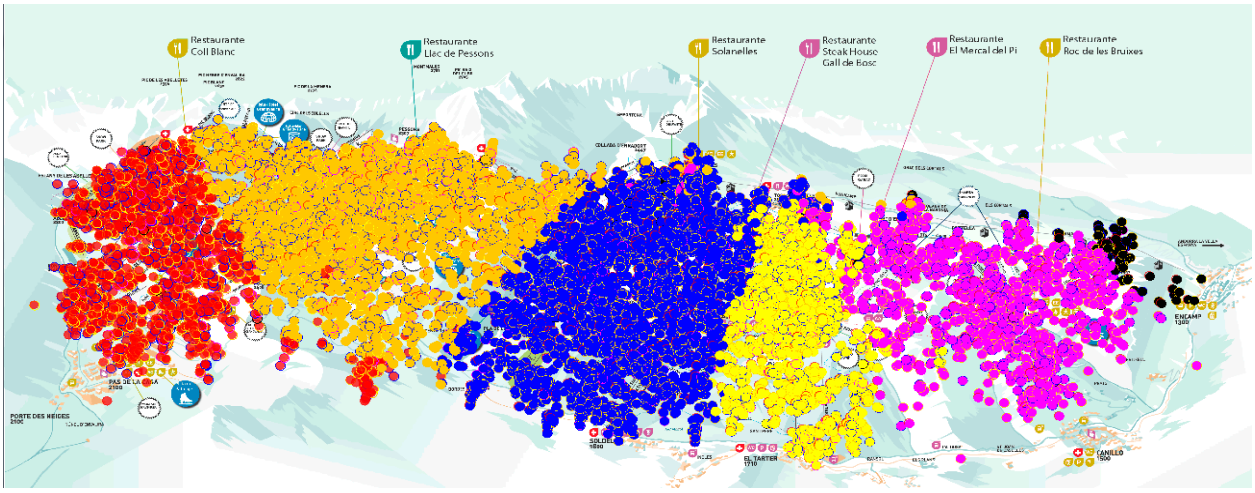

FIGURA 4.12 - SEGUNDA EXTRAPOLACIÓN DE PUNTOS, CON UN TOTAL DE 24660 PUNTOS

Los diferentes colores de los puntos representan las diferentes zonas.

4.5.4 MÉTODO Y RESULTADOS

La forma de obtener un algoritmo de reglas basada en M5 en Weka se hace separando los datos de cada coordenada que se quiere conseguir. Así, para obtener la coordenada 'x' del plano, separamos los datos así:

```
42.540919, 1.732477, 303
```

Siendo los datos: latitud, longitud y coordenada 'x'. Utilizamos las 24660 entradas como entrenamiento y obtenemos un sistema de reglas que convierte coordenadas geográficas a la coordenada 'x'.

Repetimos el proceso para la coordenada 'x' y para la zona del mapa. En este punto tenemos tres algoritmos (aunque nos centraremos solo en 'x' e 'y') para convertir coordenadas geográficas a puntos del plano.

El resultado es de este tipo:

```
Point p = new Point();

if (longitud <= 1.685 && longitud > 1.656 && longitud > 1.669 && longitud <= 1.677)
{

	p.x = (int) (3130.9169 * latitud - 22748.2566 * longitud - 93340.5181);

} else if (longitud > 1.678 && longitud <= 1.712 && longitud > 1.693 && longitud <=
1.703) {

	p.x = (int) (8571.0012 * latitud - 20280.1879 * longitud - 328975.868);

} else if (longitud > 1.678 && longitud > 1.698 && longitud <= 1.723) {

	p.x = (int) (6072.2876 * latitud - 20294.3848 * longitud - 222660.1194);

} else if (longitud <= 1.708 && longitud <= 1.673 && longitud > 1.639 && longitud >
1.653 && longitud > 1.662) {

	p.x = (int) (3181.017 * latitud - 28697.1605 * longitud - 85539.4553);

} else if (…) {

	…
}


if (latitud <= 42.555 && latitud > 42.53 && latitud <= 42.534 && longitud > 1.693 &&
longitud <= 1.726
			&& latitud <= 42.531) {

	p.y = (int) (40380.2505 * latitud + 411.0036 * longitud - 1717368.3054);

} else if (latitud <= 42.555 && latitud > 42.531 && latitud <= 42.536 && longitud >
1.692 && latitud <= 42.534
			&& longitud <= 1.727 && longitud > 1.701) {

	p.y = (int) (27038.5475 * latitud + 682.9036 * longitud - 1150397.2109);

} else if (latitud > 42.534 && latitud <= 42.559 && longitud <= 1.688 && latitud <=
42.546 && longitud <= 1.649
			&& longitud <= 1.625) {

	p.y = (int) (16477.9609 * latitud - 5900.3292 * longitud - 690620.557);

} else if (…) {
	…
}
```

FIGURA 4.13 – RESULTADO DEL PROCESO DE COONVERSION DE COORDENADAS

Como podemos ver, se trata solo de una combinación de sentencias *ifs* que comparan coordenadas. Cuando localizan el caso en el que encajan aplican una operación lineal de multiplicaciones, sumas y restas.

Hay un salto enorme entre la eficiencia del sistema original y el nuevo. La complejidad pasa de ser $O(n^3)$ a $O(1)$.

El precio a pagar es un posible error en la predicción de los puntos en el mapa. El error medio es de unos 15 pixeles y hay que tener en cuenta que el ancho del plano de Grand Valira es de unos 4000 pixels. Un error de 15 pixeles en este caso, supone un error de escasos metros en la realidad, similar al error de precisión del GPS.

4.6 INTERFAZ

4.6.1 CO-16: MAPAVISTA

Para el mapa necesitamos una vista que nos permita hacer zoom sobre la imagen con dos dedos. Android no proporciona ninguna vista que cumpla estos requisitos pero han sido desarrollados por terceros. En nuestro caso utilizamos la vista ImageTouchView [2]. Nos permite mostrar una imagen, el mapa, y movernos por la misma mediante gestos, ampliando, reduciendo y desplazando la misma. Ha sido necesario modificar el código para obtener una posición relativa del plano con respecto a la pantalla. Necesitamos saber qué relación hay entre la imagen y la pantalla para ver que se está viendo, y poder dibujar el resto de vistas de manera acorde.

MapaVista recibe las actualizaciones en caso de que haya un cambio en la imagen (se ha desplazado o cambiado el zoom sobre la misma) y mueve el resto de vistas en función de este cambio. Así si el usuario ha movido el mapa hacia la izquierda, todas las vistas que se dibujan sobre él tienen que moverse exactamente lo mismo.

ImageViewTouch extiende a ImageView (el Widget incluido en Android para mostrar imágenes) y nos permite incluir la funcionalidad de hacer zoom y movernos utilizando gestos. Nuestra adaptación del código consiste en capturar esos cambios y establecer en un rectángulo la posición relativa del mapa respecto a la vista. Con cada cambio se envía ese rectángulo al componente CO-05 MapaActivity para que represente el resto de elementos encima de esta vista.

4.6.2 CO-17: THEMISSINGTABHOST

Este componente prácticamente no hemos necesitado modificarlo. El único pequeño cambio que hemos hecho es quitar el texto que aparece en las pestañas al poner el móvil en posición apaisada, para dejar mayor espacio a las actividades.

4.7 COMUNICACIONES

En este apartado vamos a centrarnos en la interconexión de las dos grandes partes, el cliente Android y el backend en App Engine, o entre el modelo y la vista y el controlador, y los mensajes que se transmiten. En el cliente Android las comunicaciones las gestiona la clase Util.Comm, mientras que al otro lado se encuentran los servicios explicados en la sección Google App Engine.

4.7.1 ASYNCTASK

Llegados a este punto, es necesario introducir un recurso ampliamente utilizado en Android. En determinadas ocasiones, tenemos programas que tardan bastante tiempo en realizar una tarea en concreto y queremos que esta no bloquee la aplicación mientras se está ejecutando.

Para solventar esto normalmente se hace uso de los Threads. Para ello, creamos un hilo adicional para nuestra tarea y continuamos con el hilo normal para el resto de la aplicación.

El problema es que la interfaz gráfica (UI) de Android no permite llamadas desde otros hilos que no sea el suyo, así que si necesitamos realizar alguna actualización (aumentar barra de porcentaje, añadir información…) nuestra aplicación se cerrará. Hay distintos métodos para poder acceder al hilo de la UI y actualizarla desde este (Handler, runOnUiThread), pero el más sencillo de implementar, seguro y que ofrece mayores garantías es ASyncTask.

Extendiendo de AsyncTask (tarea asíncrona) podemos crear una clase que tenga entre 1 y 4 métodos según necesitemos. Además, la clase AsyncTask dispone de tres tipos de parámetros distintos, que tendremos que especificar cuándo declaremos nuestra clase.

```
public class MiTarea extends AsyncTask<Params, Progress, Result> {
}
```

Los tres parámetros son los siguientes:

- Params: Datos que pasaremos al comenzar la tarea
- Progress: Parámetros que necesitaremos para actualizar la UI.
- Result: Dato que devolveremos una vez terminada la tarea.

Una tarea asíncrona que no se debe ejecutar en el hilo principal, para utilizarla como ejemplo, es cualquiera que tenga que realizar una conexión con el servidor. Supongamos que como parámetro recibe la dirección como un String, actualiza el progreso mediante un Float y devuelve un boolean para indicar si ha tenido éxito o fracasado:

```
public class MiTarea extends AsyncTask<String, Float, Boolean> {
}
```

Si alguno de los parámetros no lo necesitamos, podremos substituirlo por Void.

Los métodos que tendremos que escribir de nuestra clase son los siguientes, aunque solo es obligatorio el primero:

```
protected Result doInBackground(Params... p) {
}
```

Este método será el encargado de realizar la tarea en segundo plano. Como vemos, recibe un número cualquiera de parámetros del tipo Params, así que debemos tratar a p como un array.

Este método se ejecuta en otro hilo, por lo que no podremos modificar la UI desde él. Para ello, usaremos los tres métodos siguientes.

```
protected void onPreExecute() {
}
```

Este método se ejecutará antes de doInBackground, por lo que podremos modificar la interfaz para indicar el comienzo de la tarea (colocar un cargando, desactivar botones…).

```
protected void onProgressUpdate (Progress... values){
}
```

Este método permitirá actualizar la interfaz mientras se ejecuta la tarea asíncrona. Para ello, desde doInBackground deberemos llamar a publishProgress y pasarle los parámetros oportunos.

```
protected void onPostExecute (Result result){
}
```

Este último método se ejecuta tras terminar doInBackground y recibe como parámetro lo que este devuelva.

4.7.2 DETALLE DE COMUNICACIONES

Como hemos dicho, la clase Util.Comm en el proyecto Android es la que implementa todas estas tareas asíncronas para facilitar la comunicación con el servidor. Vamos revisar detalladamente cada una de ellas.

4.7.2.1 CO-18: REGISTRO Y ENVIACONTACTOS

El registro con un servicio de Google App Engine requiere seguir una serie de pasos. Existen varios ejemplos en la documentación que nos facilitan la tarea pero aun así es necesario entender bien que hace el código que utilizamos.

El proceso inicia extrayendo un token del gestor de cuentas Android. Este token se envía al servidor y se recibe una Cookie que usaremos como autentificación.

Esta cookie se usara posteriormente en el resto de Componentes de comunicaciones (CO-19 a CO-22).

Este componente también registra el dispositivo con los servidores de Google para poder recibir mensajes push C2DM.

La parte de enviarContactos funciona basándose en el componente CO-10. Se obtiene un objeto Contactos que es el encargado de recorrer la agenda y devolver los teléfonos y emails que aún no se han comprobado. El método enviarContactos dispone de un Receiver que se encarga de gestionar la respuesta en caso positivo o negativo, mostrando un mensaje en tal caso.

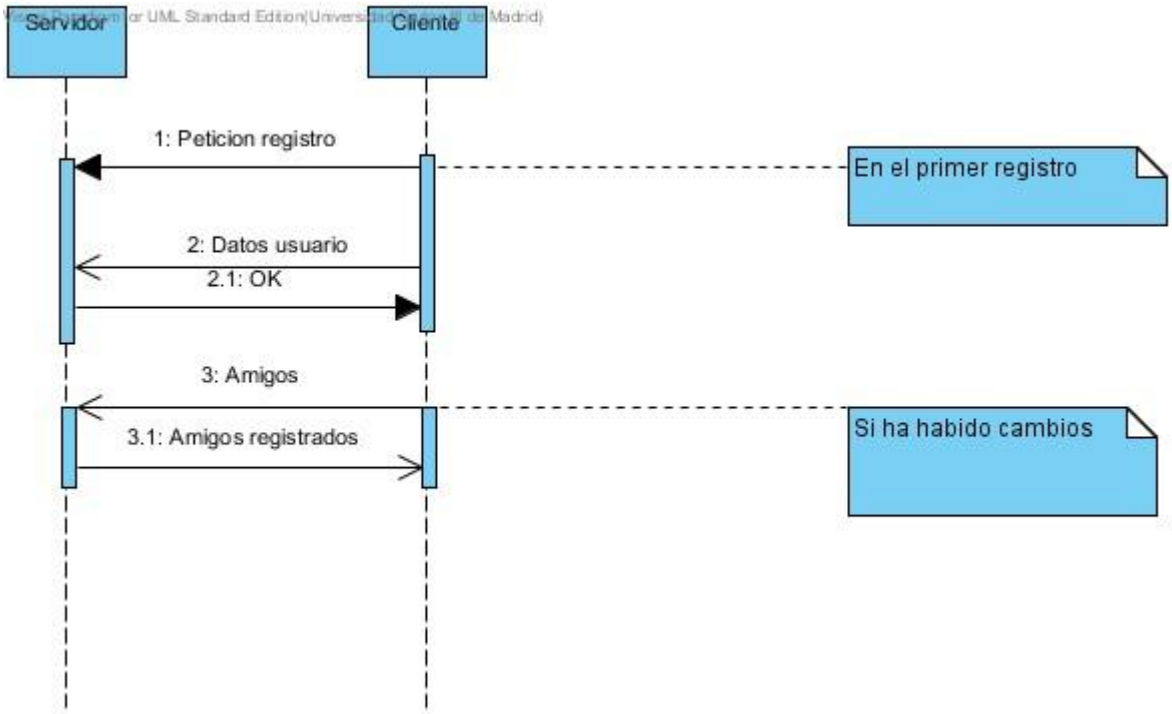

FIGURA 4.14 - PROCESO DE REGISTRO Y BÚSQUEDA DE AMIGOS

4.7.2.2 CO-19: ENVIARPOSICION

La implementación de este método es sencilla. Se pide una instancia de RequestFactory utilizando la Cookie obtenida en el registro y se inicializa.

Utilizamos el RequestFactory para obtener una instancia de un PosicionRequest. Se explica este procedimiento en la sección de Google App Engine en Diseño. Después solo hay que pedir un objeto de tipo PosicionProxy al PosicionRequest y establecer las coordenadas. Este PosicionProxy se utiliza para llamar al método `registraPosicion` en PosicionRequest. Hay en un listener que se ejecutan de forma asíncrona en caso de éxito o fracaso.

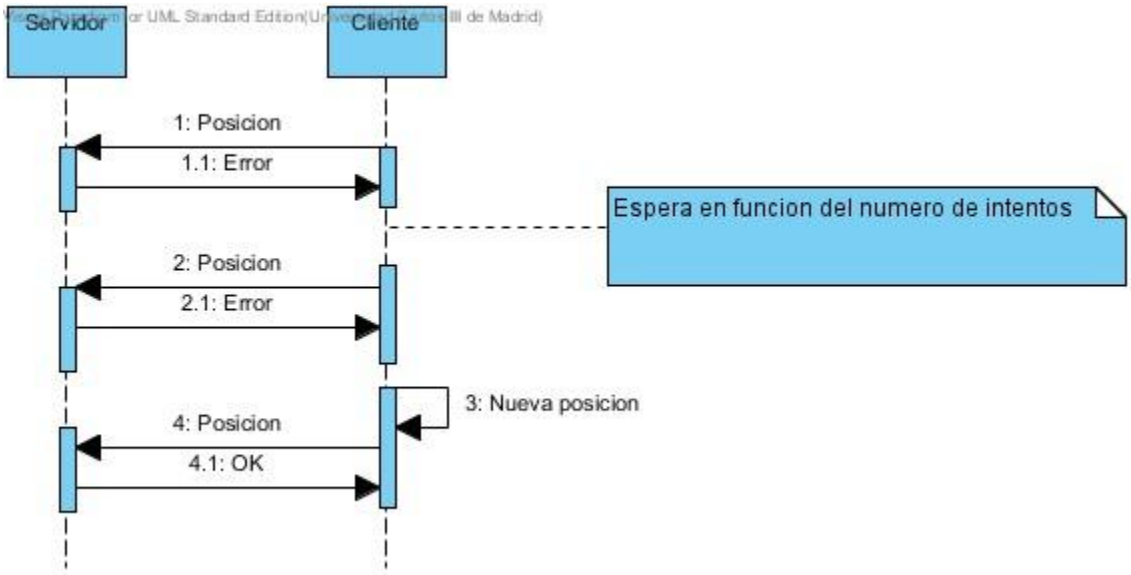

FIGURA 4.15 - PROCESO DE ENVÍO DE POSICIÓN

4.7.2.3 CO-20: CONSULTAPOSICIONAMIGOS

En primer lugar obtenemos un objeto PosicionRequest de forma análoga al componente anterior. Después disparamos el método `consultaPosicionAmigos` en PosicionRequest. Se captura el evento al finalizar de forma asíncrona y en caso de éxito, el servidor devolverá una lista de los contactos que usan la aplicación. Esta lista de contactos se pasa al componente encargado de gestionar la base de datos (CO-09) que la almacena.

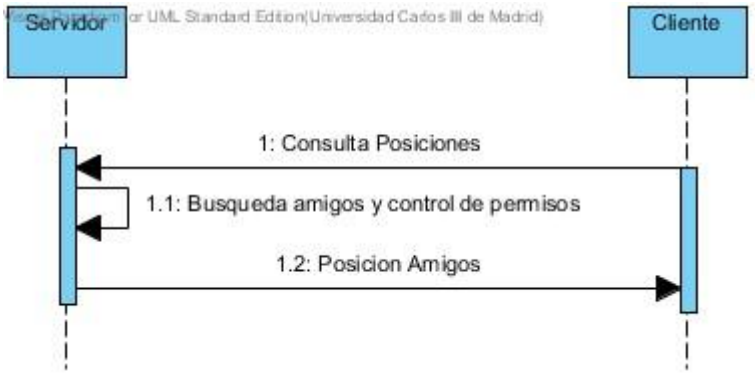

FIGURA 4.16 - PROCESO DE CONSULTA DE POSICIONES DE AMIGOS

4.7.2.4 CO-21: CONSULTAPERMITIDOS Y ESTABLECEPERMISO

Consulta al servidor el estado de los permisos concedidos. A través de la interfaz, el cliente Android establece permisos mediante establecePermiso. El servidor responde en caso de éxito o fracaso. Esto provoca que la configuración de permisos de cada usuario se encuentre tanto en el cliente como en el servidor, por lo que podrían darse estados inconsistentes. Para evitar esto el cliente llama a consultaPermitidos de forma asíncrona cuando se accede a la configuración de permisos en el cliente y sincroniza ambos estados.

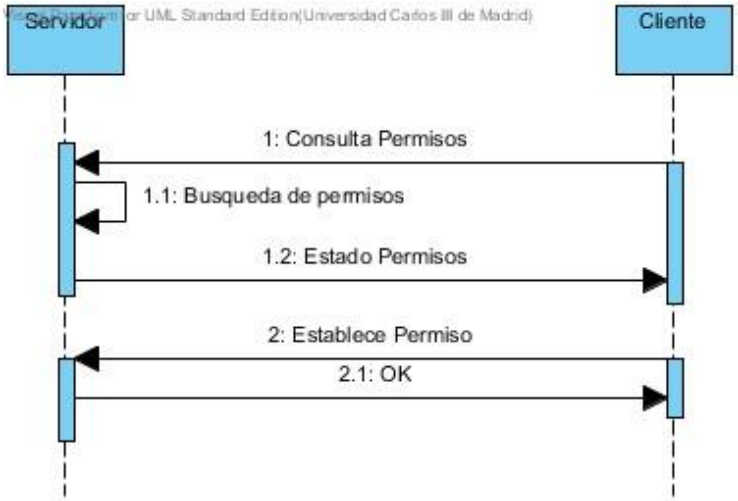

FIGURA 4.17 - PROCESO DE ESTABLECIMIENTO Y CONSULTA DE PERMISOS

4.7.2.5 CO-22: HACERPING Y POSICIONDISPONIBLE

El proceso implementado es el siguiente:

- Un usuario hace Ping, se llama al método hacerPing en el cliente, con el email del destinatario.
- El servidor recibe la petición a través del servicio de Posiciones y envía un mensaje C2DM al destinatario. Hay un BroadcastReceiver encargado de recibirlo y continuar el proceso.
- El destinatario lo recibe e inicia el servicio de posicionamiento. Si ya estaba activado, pide una nueva posición. Si no lo estaba, se activa, se obtiene una posición (o se cancela por timeout) y se desactiva el servicio, como se especifica en el componente CO-12.
- El destinatario envía la posición al servidor e indica a que usuario hay que notificar (el usuario que inicio el Ping). Puede darse el caso que más de un usuario este esperando la posición del destinatario y que se notifique a varios.
- El servidor recibe la posición y avisa al emisor o emisores que la posición pedida está disponible.
- El emisor consulta la posición al servidor

Se detalla el proceso en el siguiente diagrama de secuencia:

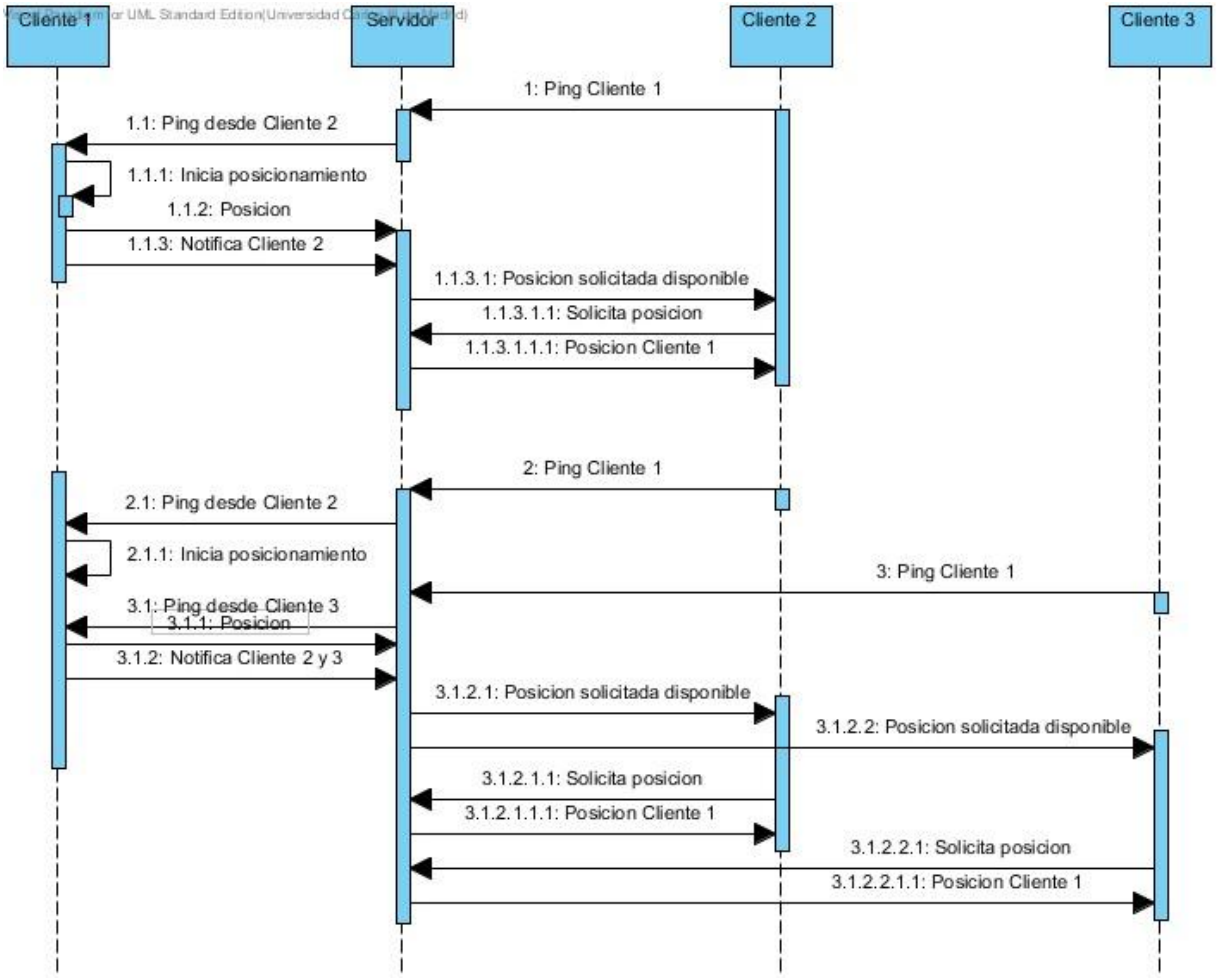

FIGURA 4.18 - PROCESO PING

4.7.3 DIAGRAMA

A continuación se incluye un diagrama que incluye las clases necesarias para establecer la comunicación con el servidor. Estas interfaces de tipo Request o Proxy, sirven para modificar los objetos correspondientes en el servidor. Más detalles en la siguiente sección.

FIGURA 4.19 - DIAGRAMA DE COMUNICACIONES

4.8 GOOGLE APP ENGINE

4.8.1 CO-23 – CO-27: USUARIO, AMIGOS, PERMITIDOS, POSICIÓN, CONTACTOS

Estos componentes representan entidades en el servidor Google App Engine. El único código a implementar en ellas son los atributos que hemos establecido en la sección Diseño y los getters y setters correspondientes. Su implementación es trivial salvo por que es necesario especificar que la clase es de tipo Entidad con el label:

```
@PersistenceCapable(identityType = IdentityType.APPLICATION, detachable = "true")
```

En cada campo indicamos que debe ser persistido:

```
    @Persistent
```

Y en el caso de la clave utilizamos:

```
    @PrimaryKey
    @Persistent (valueStrategy = IdGeneratorStrategy.IDENTITY)
```

4.8.2 CO-28: REGISTROREQUEST

El mecanismo del funcionamiento de este componente es sencillo, los clientes piden realizar diferentes tipos de transacciones sobre las entidades, almacenadas en un DataStore.

Este proceso se consigue a través de un sistema de llamadas de procedimiento remoto (RPC, en sus siglas en inglés) llamado RequestFactory [8], incluido en los servicios de Google Web Toolkit. Con este sistema frontend y backend comparten una interfaz, que permite al primero operar sobre las entidades de forma prácticamente transparente.

En el servidor se crean una serie de servicios que se encapsulan en una interfaz de tipo RequestFactory. El backend se sube a los servidores de Google, donde se les asigna una dirección web. Esta dirección se configura en el cliente y tras unos cuantos parámetros más se pueden realizar llamadas a los servicios de la siguiente forma:

- Se crea un objeto de tipo `MyRequestFactory`
- Se llama a los métodos que se deseen en el objeto anterior o bien lo utilizamos para crear nuevas entidades.
- Se llama al método fire() para provocar los cambios en el servidor. Este método recibe como parámetro un Receiver que se encarga de recibir la respuesta del servidor de manera asíncrona, y que incluye métodos que se ejecutan en caso de éxito o fracaso.

De esta manera tenemos un sistema que nos permite realizar llamadas a métodos que se ejecutan en el servidor y que operan directamente sobre las entidades.

En el caso de este componente, se trata de una interfaz con los siguientes métodos:

```
@ServiceName(value = "com.skitracker.server.RegistroService", locator =
"com.skitracker.server.RegistroServiceLocator")
public interface RegistroRequest extends RequestContext {

	Request<UsuarioProxy> registraUsuario(UsuarioProxy usuario);

	Request<List<UsuarioProxy>> recuperaAmigos();

	Request<List<UsuarioProxy>> recuperaPermitidos();

	Request<List<UsuarioProxy>> recibeAmigos(Set<String> emails, Set<String> telefonos);

	Request<Void> recibePermitido(String email, boolean permitido);

}
```

Como decíamos antes, esta interfaz es común en servidor y cliente. Los métodos realizan las siguientes funciones:

- `registraUsuario`: como su nombre indica, permite a un usuario darse de alta en la aplicación. Debe estar ya autentificado con su cuenta de email e incluir un número de teléfono válido en el formato internacional. Este método se encarga de registrar y habilitar en el servidor el servicio C2DM, permitiendo al usuario recibir mensajes que implementan otros servicios que se detallan en este documento.
- `recuperaAmigos`: este método es llamado por usuarios registrados y les permite conocer una lista de los usuarios que utilizan la aplicación.
- `recuperaPermitidos`: este método indica al usuario a que contactos se les está dando permiso para conocer su posición. El resultado de llamar a este método debe ser consistente con el estado interno del cliente
- `recibeAmigos`: este método se llama tras producirse el registro con éxito de un usuario y sirve para que este pueda enviar una lista de emails y otra de teléfonos encontrada en la agenda. Guarda las listas recibidas en el DataStore en una entidad de tipo `Contacto`. Este método devuelve una lista con los contactos conocidos, al igual que recuperaAmigos. Además cumple una función adicional, si dentro de los contactos encontramos a otro usuario, se notifica a este por C2DM de que existe un nuevo usuario que le tiene como Amigo. Posteriormente, el usuario puede forzar llamadas a este método para encontrar amigos nuevos (aunque la aplicación está pensada para que aparezcan automáticamente, como se indica en la sección Procesos Automáticos).
- `recibePermitido`: este método sirve para que el usuario pueda establecer permisos, indicando que contactos pueden ver su posición y quiénes no. El contacto se identifica por email y el segundo parámetro es de tipo boolean y establece el estado como permitido o no.

Para persistir las entidades que gestiona este servicio se hace uso del servicio de Datastore que presta Google App Engine. Vamos a verlo con un ejemplo:

```
public Posicion registraPosicion(Posicion posicion) {

	PersistenceManager pm = PMF.get().getPersistenceManager();

	// guardamos la nueva posicion
	try {
		posicion.setEmail(getUserEmail());
		posicion.setTiempo(new Date());
		pm.makePersistent(posicion);

		return posicion;
```

```
        } finally {
              pm.close();
        }

  }
```

Este es un método real, implementado en la clase PosicionService, que se encarga simplemente de recibir un objeto posición y guardarlo en el almacen de datos.

Para recuperar un objeto hacemos:

```
Posicion p = (Posicion) pm.getObjectById(Posicion.class,

KeyFactory.createKey(Posicion.class.getSimpleName(), k.getName()));
```

Esta es la forma en que todas las entidades descritas anteriormente mantienen la persistencia y son recuperadas.

4.8.3 CO-29: POSICIONREQUEST

Se trata de una interfaz con los siguientes métodos:

```
@ServiceName(value = "com.skitracker.server.PosicionService", locator =
"com.skitracker.server.PosicionServiceLocator")
public interface PosicionRequest extends RequestContext {
  Request<List<PosicionProxy>> consultaPosicionAmigos();

  Request<PosicionProxy> registraPosicion(PosicionProxy posicion);

  Request<Void> hacerPing(String email);

  Request<Void> posicionDisponible(List<String> emails);
}
```

Los métodos realizan las siguientes funciones:

- `consultaPosicionAmigos`: De carácter autodescriptivo. Consulta al servidor las posiciones de todos los contactos que hayan dado permiso al usuario, en una lista.
- `registraPosicion` : Permite al usuario indicar donde se encuentra. Se registra la posición mediante las coordenadas de latitud y longitud y se registra la hora, para una posterior consulta de otros usuarios.
- `hacerPing` : Permite hacer Ping a otro usuario. Se puede consultar su utilidad en la sección de Servicio y Posicionamiento. A nivel interno, este método envía un mensaje push C2DM a otro usuario para que este envíe su posición.
- `posicionDisponible`: Es un método que complementa la funcionalidad de hacer Ping iniciada por el método anterior. Cuando un usuario recibe un Ping, o varios de diferentes usuarios, almacena en una Colección la lista de usuarios que están esperando su posición. Cuando este tiene una nueva posición disponible, la envía al servidor mediante `registraPosicion` , y posteriormente llama a `posicionDisponible` indicando la lista de usuarios que están esperando la posición. Este método envía un mensaje mediante C2DM a cada usuario de la lista, para decirles que la posición que habían solicitado ya está disponible. El proceso finaliza cuando el usuario que comenzó el Ping llama a `consultaPosicionAmigos` y obtiene la posición solicitada. Se detalla este proceso en la siguiente sección.

La implementación del DataStore se hace de forma análoga al componente anterior.

5 GESTIÓN DEL PROYECTO

5.1 PLANIFICACIÓN DEL PROYECTO

Este proyecto se ha planificado y desarrollado como un proyecto real. Como se ha visto en las secciones de este documento, las fases del desarrollo han sido:

- Análisis
- Diseño
- Implementación

Además se han realizado numerosas pruebas aunque no estén aquí documentadas. Detallaremos los resultados en la sección de conclusiones.

En la fase de Análisis se han estudiado las alternativas tecnológicas disponibles para la realización del proyecto y se han establecido los requisitos en que se basa el resto de secciones. En la fase de Diseño, se crean los componentes de los que se compondrá en sistema. En la fase de Implementación se crea el sistema y en este documento se detalla el resultado del mismo.

La duración total del proyecto se ha planificado para durar 4 meses, distribuidos así:

- Análisis: 1-2 semanas
- Diseño: 2 semanas
- Implementación: 6-7 semanas
- Pruebas: 1-2 semanas

Aun así no se ha seguido un ciclo de desarrollo en cascada puro, sino iterativo, por lo que es difícil determinar el tiempo exacto dedicado a cada parte. En cambio, el tiempo total sí que se ajusta a los 4 meses planificados.

La redacción de este documento se ha compaginado con el resto de fases.

5.2 MEDIOS TÉCNICOS EMPLEADOS

Los medios técnicos empleados son sencillos.

En cuanto a hardware, simplemente hemos necesitado dos ordenadores portátiles Intel Dual Core, 4GB de RAM y un teléfono Android Samsung Galaxy Ace y un Samsung Galaxy S. La gran ventaja de utilizar Google App Engine es que nos podemos olvidar de alquilar y configurar servidores. Podríamos incluso haber prescindido del teléfono móvil y utilizar el emulador incluido en el SDK.

En cuando a software necesitamos más cosas, por suerte la gran mayoría son gratuitas:

- Windows 7
- Eclipse 3.7
- Subversion
- Android ADT
- Android Plugin
- Google App Engine Plugin
- Google Web Toolkit Plugin
- Drivers USB
- Microsoft Office 2010

Toda la parte de desarrollo es gratuita y el resto (Windows y Office) podrían haber sido reemplazadas por soluciones de código abierto (Linux y OpenOffice).

5.3 ANÁLISIS ECONÓMICO

Vamos a estimar el coste aproximado del proyecto. Tendremos en cuenta tanto factores humanos como recursos utilizados.

Primero analizamos el coste del personal:

Puesto	Coste / Hora	Horas	Seguridad Social [10]	Total
Desarrollador 1	15 €	160 h/mes * 4 meses	2400,00 €/mes	12.000,00 €
Desarrollador 2	15 €	160 h/mes * 4 meses	2400,00 €/mes	12.000,00 €
Total costes recursos humanos				**24.000,00 €**

El coste de hardware:

Elemento	Coste
Acer Aspire S6130	570,00 €
Sony Vaio	785,00 €
Samsung Galaxy Ace	160,00 €
Samsung Galaxy S	200,00 €
Total hardware	**1715,00 €**

No incluimos detalles del coste del software ya que las licencias de Windows y Office están incluidas en el precio de los ordenadores y el resto del SDK y demás componentes es gratuito.

Existen otros gastos indirectos que debemos tomar en cuenta. Los detallamos en la siguiente tabla.

Elemento	Coste
Consumo electricidad	100,00 €
Consumo Internet	80,00 €
Total otros	**180,00 €**

Los valores son aproximados, para el periodo de duración de 4 meses.

Vamos a analizar el coste total del proyecto:

Elemento	Coste
Recursos humanos	24.000,00 €
Hardware	1.715,00 €
Otros	180,00 €
Total antes de IVA	**25.895,00 €**
18% IVA	4.661,00 €
Total	**30.556,10 €**

Por lo tanto el coste total del proyecto se estima en 30.556,10 €.

6 CONCLUSIONES

La aplicación funciona bien. Hemos hecho bastantes pruebas individuales de cada componente por separado y todos son satisfactorios. También hemos ido a 2 estaciones de esquí y probado el sistema de localización y la compartición de posiciones con los amigos. El sistema de conversión de coordenadas funciona muy bien, y el margen de error que incluye es inapreciable para el usuario. La parte de compartir la posición con los amigos funciona bien pero en el futuro podría ser mejorada. Se puede mejorar el sistema de comunicaciones, sobre todo cuando la red no está disponible. Es crucial que esta parte funcione bien porque es la esencia de la aplicación y el motivo por el que se decidió crear una alternativa a Google Latitude.

La idea de crear esta aplicación surgió de una necesidad real, o más bien de un deseo real. Estábamos esquiando en Formigal y el grupo se separó. Es bastante habitual que siendo varios, alguno coja un remonte tarde o se retrase esquiando o se equivoque de camino. En estos casos la solución siempre es la misma, parar, coger el móvil y llamar. En una de estas ocasiones alguien dijo: ¿Y por qué no lo miras en “Latitude”? Pensé: “Que buena idea”. El problema de Google Latitude es la falta de sincronía, puede devolver posiciones de los contactos de hace varias horas. Además estando en plena montaña, sin calles, ni carreteras, ni edificios en Google Maps solo se ve un plano de color verde y nada más. Utilizando la vista por satélite tampoco se podían distinguir remontes con lo que el resultado global es que Latitude resultaba completamente inútil. De ahí surgió la idea de hacer esta aplicación orientada al entorno de la nieve.

También dudamos mucho entre utilizar Google Maps como base o crear nuestros propios mapas. Siempre con el contexto de la nieve en la cabeza pensamos que lo óptimo era utilizar los propios mapas de las estaciones, fácilmente reconocibles e intuitivos. Esta decisión también

ha supuesto el mayor reto y quebradero de cabeza pues adaptar las coordenadas geográficas a los planos personalizados (con los escasos recursos de un dispositivo móvil) ha sido la parte más difícil del proyecto pero a la vez en la que hemos encontrado una solución más creativa y que más nos satisface.

La siguiente decisión como muy bien apuntó nuestro tutor del proyecto Jorge Blasco Alís, fue la de no limitarnos a un único contexto (el del esquí) sino orientar la aplicación a convertirse en realidad en una factoría de aplicaciones. Tenía mucho sentido, pues el sistema de mapas que hemos creado permite utilizar cualquier imagen como base y superponer los elementos que nos hagan falta encima.

Sobre programar en Android, teníamos poca o ninguna experiencia. Hemos aprendido sobre la marcha solucionando los problemas como iban surgiendo y pasando horas en StackOverflow [11]. Viniendo de programar mucho en Java, el cambio a Android no es demasiado traumático pero si hay unas cuantas cosas a tener en cuenta. Android proporciona una buena base ya implementada para cada aplicación, como es el sistema de actividades, tareas asíncronas, paquetes de traducción de la aplicación a otros idiomas, vistas en XML, menús y una infinidad más de recursos que no hace falta especificar. Como Android proporciona tanto, lleva un tiempo conocerlo todo. Lo bueno de hacer esta aplicación es que nos ha obligado a tocar un número elevadísimo de las funciones que presta Android, desde crear nuestra propia vista para representar el mapa a conectarnos con servidores Google para obtener acceso a la mensajería C2DM o programar nuestro propio servicio web.

En el futuro también es necesario desarrollar una versión para iPhone. Lo bueno es que ya tenemos experiencia, el servicio web no hay que tocarlo y el diseño de los componentes nos vale en buena medida para adaptarlo a iOS.

7 REFERENCIAS

[0] Open Handset Alliance

http://www.ibm.com/developerworks/opensource/library/os-android-devel/#resources

[1] Introduction to Android development

http://www.ibm.com/developerworks/opensource/library/os-android-devel/

[2] ImageView Zoom and Scroll

http://blog.sephiroth.it/2011/04/04/imageview-zoom-and-scroll/

[3] themissingtabwidget

http://code.google.com/p/themissingtabwidget/

[4] ¿Qué es Google App Engine?

http://code.google.com/intl/es/appengine/docs/whatisgoogleappengine.html

[5] Java Servlet Technology Overview

http://www.oracle.com/technetwork/java/javaee/servlet/index.html

[6] La clase Key de Google AppEngine

https://developers.google.com/appengine/docs/python/datastore/keyclass?hl=es

7 REFERENCIAS

[7] Data-oriented Design

http://gamesfromwithin.com/data-oriented-design

[8] Getting Started with RequestFactory

https://developers.google.com/web-toolkit/doc/latest/DevGuideRequestFactory

[9] Weka

http://www.cs.waikato.ac.nz/ml/weka/

[10] Ministerio de Trabajo e Inmigración – Seguridad Social

http://www.seg-social.es

[11] StackOverflow

http://stackoverflow.com/

[12] Wikiloc

http://es.wikiloc.com/wikiloc/home.do